2010-12

An application for normal and critical operations in a tactical MLS system

Ng, Yeow Cheng.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

AN APPLICATION FOR NORMAL AND CRITICAL OPERATIONS IN A TACTICAL MLS SYSTEM

by

Yeow Cheng Ng

December 2010

Thesis Co-Advisors: Cynthia E. Irvine
Mark Gondree

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB no. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2010	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE An Application for Normal and Critical Operations in a Tactical MLS System		**5. FUNDING NUMBERS**	
6. AUTHOR(S) Yeow Cheng Ng			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**	
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**	
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number: N/A.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**	

13. ABSTRACT (maximum 200 words)

The ability for first responders to access sensitive and critical information during an emergency can help save lives and reduce damage. There may be information normally unavailable to first responders that could help during a crisis. The Transient Tactical Access to Sensitive Information (T-TASI) system is intended to employ an emergency access control policy and be a scalable security solution for transient trust. Built on a least privilege separation kernel (LPSK), the T-TASI system allows a coordinating authority to provide temporary, controlled access to sensitive information to authorized first responders, during emergencies. The current T-TASI system prototype, however, lacks applications demonstrating this capability. This work has developed a T-TASI system application. Through analysis, three necessary software subsystems were identified: a memory management system, a file storage system and an application-level library providing interfaces compliant with the standard C library. We describe the design, implementation, and testing of the application and the three supporting components, all of which will facilitate future application development for the T-TASI system.

14. SUBJECT TERMS TCX, LPSK, Partition, Emergency, Transient Trust, T-TASI, File System, Memory Management, C Library			**15. NUMBER OF PAGES** 181
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AN APPLICATION FOR NORMAL AND CRITICAL OPERATIONS IN A TACTICAL MLS SYSTEM

Yeow Cheng Ng
Civilian, Defence Science & Technology Agency, Singapore
B.Sci., National University of Singapore, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2010

Author: Yeow Cheng Ng

Approved by: Cynthia E. Irvine
Thesis Co-Advisor

Mark Gondree
Thesis Co-Advisor

Peter J. Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The ability for first responders to access sensitive and critical information during an emergency can help save lives and reduce damage. There may be information normally unavailable to first responders that could help during a crisis. The Transient Tactical Access to Sensitive Information (T-TASI) system is intended to employ an emergency access control policy and be a scalable security solution for transient trust. Built on a least privilege separation kernel (LPSK), the T-TASI system allows a coordinating authority to provide temporary, controlled access to sensitive information to authorized first responders, during emergencies. The current T-TASI system prototype, however, lacks applications demonstrating this capability. This work has developed a T-TASI system application. Through analysis, three necessary software subsystems were identified: a memory management system, a file storage system and an application-level library providing interfaces compliant with the standard C library. We describe the design, implementation, and testing of the application and the three supporting components, all of which will facilitate future application development for the T-TASI system.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

CC	Common Criteria
DSEG	Data Segment
EAL	Evaluation Assurance Level
EAP	Extraordinary Access Partition
FAT	File Allocation Table
LDT	Local Descriptor Table
LPSK	Least Privilege Separation Kernel
MLS	Multi-Level Secure
MMU	Memory Management Unit
MSEG	Memory Segment
PL	Privilege Level
RAM	Random Access Memory
SAK	Secure Attention Key
SKPP	Separation Kernel Protection Profile
T-TASI	Transient Tactical Access to Sensitive Information
TCB	Trusted Computing Base
TCX	Trusted Computing Exemplar
TPA	Trusted Path Application
TSL	Trusted Services Layer
VM	Virtual Machine

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to express my gratitude to all those who have made this thesis possible. My deepest gratitude to both my thesis advisors, Professor Cynthia Irvine and Dr. Mark Gondree, for their patience and guidance through the course of this thesis. I would like to take the opportunity to thank David Shifflett and Paul Clark who spent their time and shared their knowledge to help me complete my thesis.

I would also like to thank my Singapore sponsor, the Defence Science & Technology Agency (DSTA), for giving me the opportunity to pursue my postgraduate studies, as well as the professors and lecturers at NPS who have taught me, and from whom I have learned a lot.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. MOTIVATION

The ability for first responders to access sensitive and critical information during an emergency can help save lives and reduce damages. This may be information normally unavailable to first responders that could help during a crisis, due either to a lack of appropriate vetting in advance or to a lack of "need-to-know." The Transient Tactical Access to Sensitive Information (T-TASI) system is intended to employ an emergency access control policy and be a scalable security system to grant extraordinary access to sensitive information. The T-TASI system allows some coordinating authority to provide temporary, controlled access to sensitive information to authorized first responders, during emergencies. The T-TASI system design is based on a least privilege separation kernel (LPSK) that provides the security policy enforcement underlying the ability to allow extraordinary access with transient trust. The current T-TASI system prototype, however, lacks applications demonstrating its capabilities. The motivation of this thesis is to develop applications that will better showcase the transient trust property, and to design an application development framework that will facilitate future development for the T-TASI system.

B. PURPOSE OF STUDY

The T-TASI system currently has a very limited set of applications demonstrating its capabilities. One objective of this research is to develop applications that will showcase the ability of the LPSK to support policy-controlled transient trust. The notional scenario for these transient-trust capabilities is that of providing first responders with temporary access to sensitive data during an emergency or other exceptional situation (which is otherwise unavailable during normal operations, under traditional MLS policies). Another objective of this research is to provide a set of common libraries to help reduce the development and maintenance effort for future T-TASI System application development.

C. THESIS ORGANIZATION

Chapter I introduces the motivation and purpose for our work. Chapter II provides background information on concepts integral to understanding our development setting and the designs we propose, including separation kernels, the LPSK, the TCX project, and the T-TASI system. Chapter III describes the objectives of our project and provides a high-level analysis of our project requirements. Out of this analysis, three main software components are identified to support our project: a memory management system, a file storage system and an application-level library providing interfaces compliant with the standard C library. Chapter IV documents the design and implementation of each of these three main components: the T-TASI Application-Level Memory Management system, the T-TASI RAM Disk File System, and the T-TASI C Library. Chapter V details the functional and exception testing performed to verify to correctness of each component. Chapter VI concludes with a discussion of problems encountered, an overview of some related work and suggestions for future work.

II. BACKGROUND

The overall objective of this research is to develop applications that can illustrate how a secure, tactical device incorporating a separation kernel can be used for extraordinary access. Before exploring the details of our work, we provide background information to give the reader the context of our research. In particular, this chapter serves to provide the reader with a basic understanding of separation kernel technology, the Least Privilege Separation Kernel (LPSK) of the Trusted Computing Exemplar (TCX) project, the current LPSK prototype, transient trust and a scenario using transient trust.

A. LEAST PRIVILEGE SEPARATION KERNEL (LPSK)

A kernel is the central part of an operating system responsible for managing resources such as memory, the processor and the devices associated with the computer. When a single computer system provides information storage and computation services to a group of applications that perhaps are associated with different users there is often a requirement to control access of these applications and users to the information contained in the system. A *security kernel* is the minimal, protected core of the operating system that implements the reference monitor abstraction [1]. The correct operation of the security kernel is sufficient to guarantee the trusted computing base (TCB) satisfies the reference monitor's security properties, including mediated access, verifiable enforcement, and tamper-proof operation [1]. This security kernel is the least common mechanism [2] necessary to implement the security policy and usually includes mechanisms for information sharing, inter-process communication, and physical resource multiplexing. Given an architecture based upon the use of TCB subsets [3], it is possible to verify that the security policy allocated to the security kernel is implemented correctly independent of other elements of the TCB.

MITRE developed the first prototype security kernel, as a government-sponsored project to prove the concept [4]. Since then, a number of proprietary operating systems

have been developed using designs based on the security kernel concept. The first commercially available operating systems incorporating security kernels were Honeywell's Secure Communication Processor (SCOMP [5]) and the Gemini Secure Operating System (GEMSOS [6]). SCOMP and GEMSOS use different approaches in their implementation. SCOMP is implemented on custom hardware optimized for security while GEMSOS is implemented on existing commercial Intel x86 hardware.

Rusby argued that distributed systems offer a natural basis for the design of secure computer systems. He proposed a new type of security kernel, known as a *separation kernel*, that is able to create the same secure environment provided by a physically distributed system in a single shared machine [7]. A separation kernel performs its role by first allocating all resources to *partitions*—the semantics of which may represent different policy equivalence classes—and, second, by controlling all interactions between the partitions. A subject in one partition cannot communicate to or influence a subject in another partition, unless it is allowed by the security policy [8].

A *least privilege* [2] *separation kernel* is a class of separation kernel that, in addition to enforcing the relationships between the policy equivalence classes, applies the security design principle of *least privilege* to the interaction between subjects and resources managed by the kernel [9]. This is done through the enforcement of both partition-to-partition and subject-to-resource policies. *Partition-to-partition* policy determines the coarse interaction of all active entities in one partition with the resources in another partition, whereas the *subject-to-resource* policy further restricts the interaction of subjects with resources in another partition on a subject-by-subject basis. A *privilege level* (PL) policy, applicable within processes and between processes, constrains a subject's access to resources based on the privilege level of both the subject and the resource. The PL policy allows subjects with higher privilege (i.e., numerically lower hardware PL number) to access resources of less privilege (i.e., numerically higher PL number).

One resource available in an LPSK system is a *segment*. A segment is a block of memory to which subjects potentially have read and write access. There are three types of

segments available in the LPSK: *default segments*, *memory segments* (mseg) and *data segments* (dseg). A default segment refers to the code, data and stack segments of an executable resource. A mseg is an Intel x86 data segment that is created and exported by the LPSK to a process' address. A dseg is an Intel x86 data segment that is created by the LPSK from a secondary storage segment in a process' address space [10].

A configuration vector is the configuration data that is parsed by the LPSK Initializer to establish both the initial secure state and the behavior of the Run-Time LPSK during an operational mode. The LPSK Configuration Tool is an off-line software tool that is used to convert a human-readable configuration vector into a binary configuration vector, and vice versa. This tool is used to specify the behavior of the LPSK and the applications executing in the partition [10].

B. TRUSTED COMPUTING EXEMPLAR (TCX) PROJECT

The Information Assurance Directorate of the United States Government published "The U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness" that defines the *Separation Kernel Protection Profile (SKPP).* This document provides the requirements for a high assurance *separation kernel* [11].

One aspect of the Trusted Computing Exemplar (TCX) project at the Naval Postgraduate School is to develop an openly-distributed separation kernel that provides high assurance for applications such as simple embedded systems [12]. The objectives of the project are to create a reusable high assurance development framework, develop a reference implementation of trusted components, support the evaluation of these reference components at the Common Criteria [13] evaluation level EAL7 and provide open dissemination of the previous three activities. The Least Privilege Separation Kernel (LPSK) developed by the TCX project is intended to be compliant with the SKPP.

C. TRANSIENT TACTICAL ACCESS TO SENSITIVE INFORMATION (T-TASI) PROJECT

The Transient Tactical Access to Sensitive Information (T-TASI) project is related to the TCX project. The T-TASI project introduces the concept of *emergency access* [12] in the context of a small tactical device (E-device) that hosts the LPSK and related trusted and untrusted services. First-responders (e.g., police, medical personnel, fire safety personnel) require information that, if made available, could assist them to better handle an emergency situation. Examples of such information include those relevant to physical security—such as infrastructure blueprints—or those relevant to private medical information. In general, if this information is made widely available in advance of any "need to know," it may lead to damage. *Transient trust* is the idea of providing access to information, in temporary violation of some alternative *normal mode* policy (i.e., *extraordinary access*), so that users can securely accomplish some immediate and necessary task [14]. *Transient trust* is extended to the user during an emergency, allowing users access to sensitive emergency information in accordance with some (permissive) *emergency mode* policy.

During normal-mode operations, the user of the E-device has no access to sensitive information. This sensitive information resides in the *Extraordinary Access Partition* (EAP) of the E-device, which may be updated in the field. The EAP is scheduled during normal mode operations but only has keyboard and screen focus (i.e., can be accessed by the user) during an emergency. In addition, the user is only able to enter the EAP during an emergency after proper identification and authorization via the *Trusted Path Application* (TPA).

D. T-TASI SYSTEM

The prototype system developed in the T-TASI project is known as the *T-TASI System*. The current prototype system consists of the LPSK kernel, Trusted Services Layer (TSL), Trusted or Untrusted Operating System Services, Trusted Path Application (TPA), partitions (e.g., EAP and normal partitions) and various applications (see Figure 1). The system prototype currently supports multiple partitions on an E-device, with one

process per partition. A variety of input and output devices are either virtualized and multiplexed to the partitions by the LPSK, or are focused on a particular partition as a result of user interaction or some system event [10].

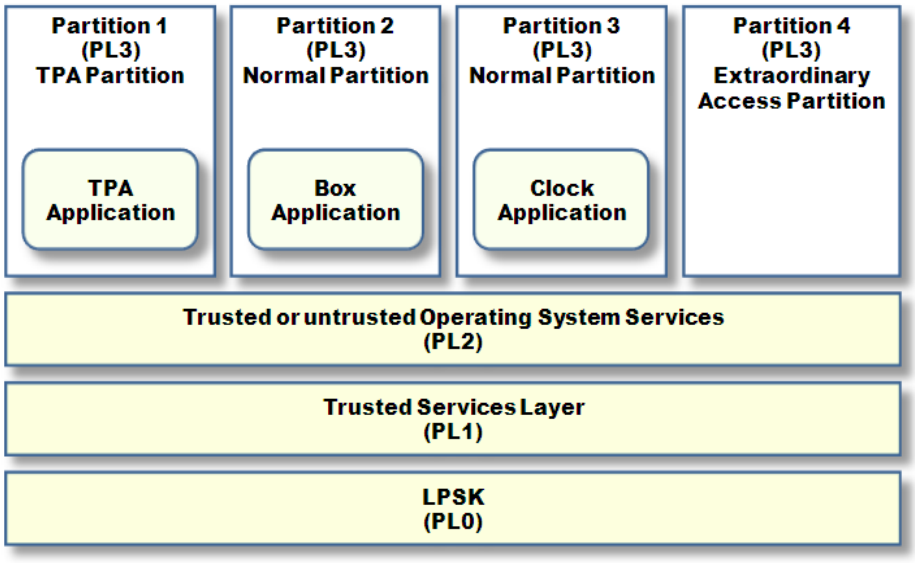

Figure 1. Components of the T-TASI System

The architecture of the T-TASI is summarized in Figure 1. The various architectural components, and their associated hardware protection level, are described next; the interested reader is referred to Irvine *et al.* [15] for details.

(PL0) The LPSK executes in PL0 and is designed to meet the security requirements of the SKPP. It provides resource partitioning and management, mandatory access control policy enforcement, process/partition scheduling, cross-partition and inter-process communication and *Secure Attention Key* (SAK) detection [16].

(PL1) The *Trusted Services Layer* (TSL) executes in PL1 and provides Multilevel Security (MLS) support and interpretation, resource virtualization, object management, focus management, trusted channel management, internet routing and inter-partition networking and emergency management.

(PL2) The *Trusted and Untrusted Operating System Services* execute at PL2 and provide application management, user management and operating system services.

(PL3) Three types of partitions are exported by the LPSK: TPA Partition (Trusted Partition), Normal Partition, and EAP. Normal partitions are used to support regular data processing activities of a user. The TPA partition provides a high assurance execution environment for high integrity applications such as the TPA. The EAP contains sensitive data that the user is not authorized to see under normal conditions and that is only available during certain emergencies. An Application I/O Library provides console input and output services at PL3 for applications, and provides some interfaces for formatting console output.

To demonstrate the current capabilities of the system, three applications have been developed: the TPA, the box application and the clock application. The box application demonstrates the console input and output capabilities of the T-TASI system by rendering random graphical boxes on the E-device screen. The clock demonstrates the timer and scheduling functions of the T-TASI system. The TPA provides a trusted path application: a trusted communications path between the user and the system for initial authentication. The TPA also provides the user with an interface to change partitions, as permitted by the user's session level and access rights. There is no existing application (in the EAP) to demonstrate extraordinary access to information during emergencies. Provision of an EAP application will be the focus of the work to be discussed in the chapters that follow.

E. SUMMARY

This chapter provided the background of Separation Kernel technology, the Least Privilege Separation Kernel (LPSK) of the Trusted Computing Exemplar (TCX) project, the T-TASI system and transient trust. It also introduced a scenario using transient trust. The next chapter describes the objectives and a high-level analysis of the requirements.

III. OBJECTIVES AND HIGH LEVEL ANALYSIS

This chapter is separated into four sections. The objectives of this research are described first, followed by a description of the concept of operations for a text editor in a scenario requiring emergency access of sensitive data. The third section provides an analysis of requirements for the text editor application.

A. OBJECTIVES

The T-TASI system currently has a very primitive set of applications demonstrating the capabilities of the LPSK. Applications for the existing T-TASI system are: a trusted path application, a clock application and a box rendering application used for terminal display functionality demonstrations. One objective of the research described here is to develop applications that will showcase the LPSK capabilities, such as the ability to support policy-controlled transient trust [15]. The notional scenario for these transient-trust capabilities is that of providing first-responders with temporary access to sensitive data, otherwise unavailable during normal operations under traditional MLS policies, during an emergency or other exceptional situation.

Utility functions, such as memory management, file I/O access and string manipulation, are traditionally provided for development by standard C library interfaces. These interfaces, however, are not available in the current T-TASI system prototype. As argued by Guillen [17], in the absence of a standard C library, development becomes expensive and support for these commonly used functions will likely be repeated and implemented on an application-by-application basis. Alternatively, using a common library reduces development and maintenance effort and is better aligned with the principle of least common mechanism [2]. This motivates a secondary objective of this research: to provide a framework for easier development of future applications for the T-TASI system.

B. SCENARIO (CONCEPT OF OPERATIONS)

During *normal mode operation*, a user (i.e., a potential first-responder) of the prototype-TASI system has no access to sensitive information. During normal mode operation, users can freely create and access information in a normal partition. Sensitive information, however, resides in the *extraordinary access partition* (EAP) of the E-device, which is not accessible to users during normal mode operation. The EAP is scheduled during normal mode operation; however, it is permitted to have keyboard and screen focus (activated) only during an emergency. When the network message indicating the start of an emergency arrives, the E-device transitions to *emergency mode*. At this point, screen and keyboard focus will change to the TPA partition and the TPA will display a message informing the user that the EAP may be accessed. The user authenticates through the TPA and selects to transfer focus to the EAP to interact with the emergency application. When the emergency is over—as before, signaled by a network message—the EAP is no longer accessible and the system will transition back to normal mode.

A text editor application allows the user to read sensitive information in the EAP that is pertinent to handling the emergency. The text editor application also allows the user to update information or store new data in the EAP. The information written to the EAP can be synchronized over the network with a main authority at a later time. Sensitive information can created in advance and stored in the EAP for use during an emergency. During emergency mode operation, a user can access sensitive information in the EAP but, due to the LPSK partition configuration policy, the user will not able to copy this information to normal partitions or to a remote site using the network. This policy prevents the leakage of sensitive information during or after the emergency period.

C. APPLICATION SELECTION AND REQUIREMENTS

A framework will be created to allow an open source text editor to be ported to the T-TASI system. Compared to application development “from scratch,” porting applications eliminates redevelopment of similar functionality and shortens the

development time. Other capabilities (i.e., applications) can be implemented on the T-TASI system by using this porting framework.

There are many open source text editor applications that are suited for this research work, such as GNU *Emacs* and other command line editors, various graphical editors and general text manipulation utilities. Table 1 shows a list of text editor applications that were originally considered and their external dependencies. Many of these applications depend on external libraries, such as the *libc* (standard C library), *libm* (math library), *pthread* (multithreading) and *ncurses* (textual user interface) libraries; however, these libraries are not available in the current T-TASI system prototype. In addition, it is not straightforward to implement many of these libraries, as they themselves require features of the LPSK and the Trusted Services Layer that are not yet available. Thus, a phased prototyping approach, as followed in the LPSK and TSL development, is also followed in our project.

Table 1. Potential Text Editor Applications

no.	Application	Description	Requirements / Dependencies
1.	GNU *Emacs* [18]	An extensible and customizable display editor	• ncurses • pthread • libm • libc
2.	*Vim* [19]	A general purpose, text-based editor	• ncurses • pthread • libm • libc
3.	*ed* [20]	A simple line editor	• libc

The GNU project's open source text editor *ed* [20] was selected as our text editor application because it appeared particularly simple to port and had less dependencies compared to other applications. The current *ed* project (version 1.4) has dependencies on functions in the standard C library. To support porting of this existing code to the T-TASI system, these missing functions have to be implemented in a customized C library for the T-TASI system. Table 2 provides a list of functions that *ed* requires from the standard C library; the name for each function is presented in the second column, the third column shows the usage classification (whether the interface is used for process handling, signal handling, file management, memory management, console interfaces or straightforward utility routines) of the interface and the fourth column provides a brief description of the interface.

Table 2. Function Dependencies of ed Application

no.	Function Name	Usage	Description
1.	*setjmp*	Process	Save the current application environment for *longjmp*
2.	*longjmp*	Process	Restores the application environment set by *setjmp*
3.	*exit*	Process	Terminate the existing application
4.	*getenv*	Process	Get the application environment of a given value
5.	*isatty*	Process	Test whether a given device is a terminal
6.	*pathconf*	Process	Get the path name configuration and limits
7.	*pclose*	Process	Close a pipe stream
8.	*popen*	Process	Open a pipe stream
9.	*setvbuf*	Process	Assign buffering to a stream
10.	*system*	Process	Issue an external command from the application
11.	*setlocale*	Process	Set the application specific locale
12.	*ioctl*	Process	Control a stream device
13.	*sigaddset*	Signal	Add a signal to a signal set
14.	*sigemptyset*	Signal	Remove a signal from a signal set
15.	*sigaction*	Signal	Specify an action to be associated with a given signal
16.	*sigprocmask*	Signal	Change a blocked signal
17.	*fclose*	File	Close an opened file

no.	Function Name	Usage	Description
18.	*fopen*	File	Open a file with a given file name
19.	*fflush*	File	Flush unwritten data in the output buffer to the file
20.	*fputc*	File	Write a character to the file
21.	*fread*	File	Read a block of data from an opened file
22.	*fseek*	File	Reposition the file stream pointer
23.	*ftell*	File	Get the current file stream pointer
24.	*fwrite*	File	Write a block of data to an opened file
25.	*stderr*	File	Standard error file descriptor
26.	*stdin*	File	Standard input file descriptor
27.	*stdout*	File	Standard output file descriptor
28.	*strerror*	File	Get an error message string
29.	*tmpfile*	File	Create a temporary file
30.	*malloc*	Memory	Allocate a block of memory
31.	*free*	Memory	Free an allocated block of memory
32.	*realloc*	Memory	Reallocate the size of a previously allocated memory block
33.	*regcomp*	Utility	Compile a given regular expression
34.	*regerror*	Utility	Provide a mapping of regular expression error codes to printable string
35.	*regexec*	Utility	Compare a given string with the regular expression

no.	Function Name	Usage	Description
36.	*regfree*	Utility	Free any memory used by *regcomp*
37.	*strtol*	Utility	Convert a string to a long integer
38.	*memchr*	Utility	Find a byte in a block of memory
39.	*memcpy*	Utility	Copy a block of memory from one location to another
40.	*strchr*	Utility	Find a character in a given string
41.	*strlen*	Utility	Find the length of a given string
42.	*strncmp*	Utility	Compare parts of two given string
43.	*strncpy*	Utility	Copy part of a string
44.	*scanf*	Console	Get a character from the console input
45.	*printf*	Console	Output formatted string to the display
46.	*puts*	Console	Output a string to the display

For any text editor application to execute (i.e., to open, read, write and save a file), the T-TASI system will need to provide some type of file storage system. This file storage system must be able to support file management functions such as file open, file read and file write. As the current platform supports multiple partitions with different privilege levels, this file system must also be able to support multiple instances in different partitions so as not to mix sensitive and non-sensitive information in the same file storage system. No such file system is currently available for the T-TASI system.

The *malloc* and *free* functions (see Table 2) are used for memory management in the *ed* application. External memory is available to an application running in a partition, and is provided through the use of memory segments exported by the LPSK to the partition. A memory management system needs to be implemented to support between these C library memory functions using the memory segments exported by the LPSK.

D. SUMMARY

This chapter motivates this work by introducing a scenario describing the use of a text editor during an emergency. The larger objectives of this work and a high level analysis of the requirements for our application were provided. The *ed* text editor was chosen as the application for this project. This analysis resulted in some preliminary rationale for developing a customized C library, a file storage system, and application-level memory management support for the T-TASI system. The next chapter will continue with the design and implementation of these components.

IV. DESIGN AND IMPLEMENTATION

This chapter is separated into eight sections: first, we present a high level design of the system, then we provide a description of the design and implementation for each of the three main components of our development framework (the T-TASI Application-Level Memory Management, the T-TASI RAM Disk File System and the T-TASI C Library); we conclude with a summary of this chapter.

A. HIGH LEVEL DESIGN

The design of the system is intended to satisfy the objectives of our project and the requirements for the *ed* application. We previously identified three main components (see Chapter III, Section C) that are required to support an application like *ed* on the T-TASI system:

1. A C library to provide a standard interface to common utility functions.
2. A memory management component to provide an interface between the C library memory functions and the memory segments exported by the LPSK to the partition. This memory management will support the dynamic allocation and de-allocation of memory of the application when it is running.
3. A file system to provide storage to allow application data to persist.

Figure 2 shows a high-level design and how these components interrelate to support the *ed* application.

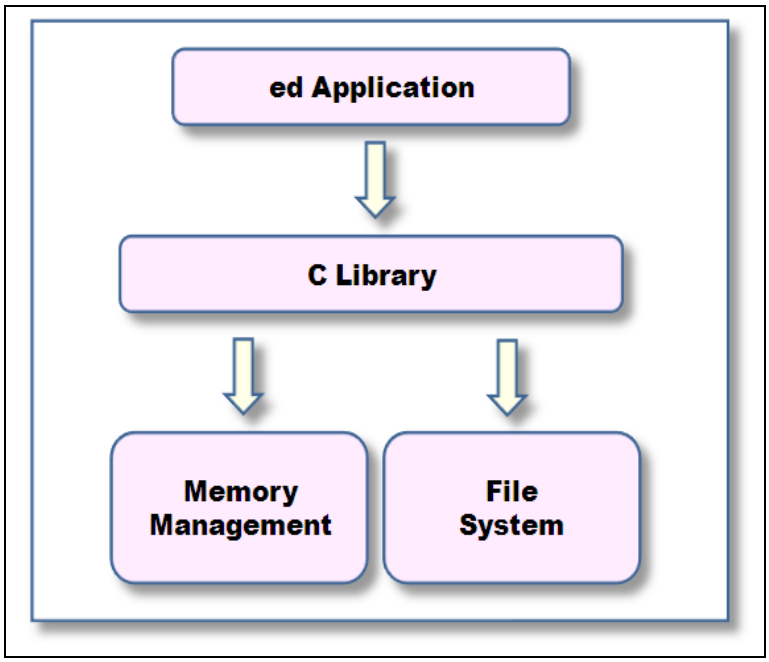

Figure 2. Dependency Relationships Among *ed* and Our Three Main Software Components

In most standard C library implementations, memory allocation and de-allocation functions invoke a kernel service that provides the memory operation. Memory allocation in the T-TASI system is static, with memory segments exported by the LPSK to the partitions during system initialization. A memory management system, the T-TASI Application-Level Memory Management component, is introduced to manage the use of the exported memory segments on behalf of applications such as those relying on the T-TASI C Library. Specifically, this memory manager will allocate memory blocks from the exported memory segments to requesting applications when the *malloc* function in the T-TASI C Library is invoked. The design of the T-TASI Application-Level Memory Management is described in Section B and its implementation is described in Section C.

A file system is introduced to provide an application-friendly storage capability in the T-TASI system. This file system, the T-TASI RAM Disk File System, will provide storage capability to applications in the partitions. Applications will access file input and output functions through T-TASI C Library interfaces such as *fopen* and *fread*. The T-TASI C Library will act as an abstraction layer, providing interfaces for applications

requiring file input and output functionality. This allows the file system to be developed and improved independently without affecting the applications. The design of the T-TASI RAM Disk File System is described in Section D and its implementation is described in Section E.

The T-TASI C Library will provide a basic C library for applications in the T-TASI system for I/O operations, memory handling and string manipulation. A common C library will reduce the complexity of future applications for the T-TASI system because developers will no longer need to implement the same functionality separately, for each application. The net effect of a common library is less development time and fewer programming errors. This approach is also aligned with the principle of least common mechanism in secure system design principles [21]. The design of the T-TASI C Library is described in Section F and its implementation is described in Section G.

B. T-TASI APPLICATION-LEVEL MEMORY MANAGEMENT DESIGN

Memory management is required to support the dynamic allocation and de-allocation of memory at the application level. The LPSK prototype exports memory segments to a particular partition using the configuration vector. In particular, the configuration vector describes the amount of memory and the allowed access modes (read, write or both) to that memory, for each partition. Once the memory allocation is defined in the configuration vector, it cannot be changed by either the kernel or applications. From a memory management perspective, this may result in unnecessary wastage of memory, i.e., an application may only require some memory for a short period of time.

The T-TASI Application-Level Memory Management module is intended to manage the memory segment allocated to a partition by the configuration vector. The memory management module will allocate portions of a memory segment to an application at its request, and free the allocated memory when the application no longer

requires it. This memory segment used by the memory management subsystem will be shared with multiple applications in the same partition. (Note that at present T-TASI system supports only one application per partition.)

1. Interfaces

T-TASI Application-Level Memory Management component will provide two interfaces for memory allocation and deallocation as shown in Table 3. The interface *get_memory* will be used for memory allocation request and *free_memory* will be used for freeing up the allocated memory.

Table 3. T-TASI Application-Level Memory Management Interfaces

no.	**Function Interface**	**Function Description**
1.	int get_memory(unsigned int size, void** ptr)	This function allocates memory to the application from the internally managed memory segment. Inputs: size [IN] Specifies the number of bytes of the requested memory allocation ptr [OUT] A pointer to the memory block allocated by the function. If the function fails to allocate the requested memory, a null pointer is returned. Function Result: Function return 0 when successful and 1 if there is an error.

no.	Function Interface	Function Description
2.	int free_memory(void* ptr)	This function frees the memory block previously allocated from *get_memory*. Inputs: ptr [IN] Pointer to a memory block to be freed. This memory block must be previously allocated with *get_memory*. Function Result: Function return 0 when successful and 1 if there is an error.

2. Dependencies

The T-TASI Application-Level Memory Management module is implemented in PL3, at the same privilege level as the application. It depends on *make_ptr*, a Trusted OS Service interface available in PL2, which is used to create a pointer to the memory segment for its memory management. The relationships between each of these components are illustrated in Figure 3.

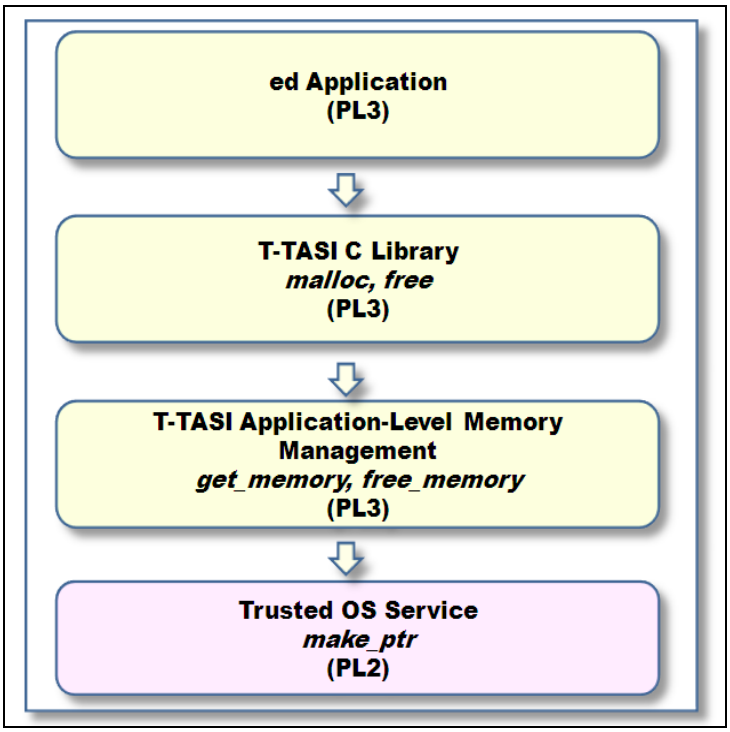

Figure 3. T-TASI Application-Level Memory Management Relationships

C. T-TASI APPLICATION-LEVEL MEMORY MANAGEMENT IMPLEMENTATION

1. Initialization

The offline LPSK Configuration Tool is used to configure the size of the memory segment used by the T-TASI Application-Level Memory Management module. The memory segment exported by the LPSK is initialized into a single memory array within the memory module. Blocks of memory are dynamically allocated to applications from this array, as memory requests are made. For each block of memory allocated for an application's use, the first five bytes are reserved for administrative (bookkeeping) purposes, and we refer to these as the memory block's an *administrative block*. The first byte of the administrative block indicates if the memory block following it is considered free (is equal to zero), or has been allocated to some applications for use (is equal to one). The next four bytes indicate the size of the memory block following the administrative

block. Administrative blocks are fixed-size, so the sixth byte is always the beginning of the memory block controlled by the administrative block.

When the *get_memory* is invoked for the first time, the T-TASI Application-Level Memory Management module will perform its initialization. During initialization, the entire internal memory array is interpreted as a single memory block. The first byte of its administrative block is set to zero, indicating the memory block is free, and the next four bytes are set to size of the memory block following the administrative block (i.e., the size of the memory segment, less five bytes).

2. Memory Allocation

When the function *get_memory* is called, the memory manager scans the array for an appropriately sized, free memory block according to a *First-Fit* allocation scheme. A First-Fit allocation scheme allocates memory using the first free memory block that is at least as large as the requested block. When a free memory block is found and the size of the free memory block is precisely the requested size, then its administrative block is modified to reflect that the block is no longer free, and *get_memory* returns success. If the size of the requested memory block is smaller than the free block, the free block will be split: the free block will be marked as allocated, but its length reduced to the requested size; a new administrative block will be created after this, marking the remaining unallocated memory as a new, free block. This first memory block will be passed to the requesting application as allocated memory. If no free memory block accommodating the allocation request is found, *get_memory* returns failure.

Bays' results [22] suggest that a First-Fit allocation scheme, as we use, has better or comparable performance to other simple allocation schemes, such as Next-Fit and Best-Fit. First-Fit is also a fast algorithm because it spends as little time searching for available memory as possible [23]. We have chosen a First-Fit allocation scheme, as it is quite simple to implement compared to other allocation schemes.

3. Memory Deallocation

When the function *free_memory* is called, the address of the allocated memory is passed as a parameter to the function. The first byte of the administrative block for this memory block is this address minus five bytes. The first byte of the administrative block will be set to free, i.e. unallocated, to indicate this memory block is no longer in use. On each de-allocation, the freed memory block will, if possible, be merged with adjacent free memory blocks. This helps to prevent fragmentation of available memory over long periods of allocation and deallocation activity. When merging two adjacent free memory blocks, the first administrative block's length is expanded, causing the second memory block and its administrative block to be recovered as available memory.

D. T-TASI RAM DISK FILE SYSTEM DESIGN

Currently, there is no secondary storage device driver available in the T-TASI system. To support the demonstration scenario, a RAM disk storage device was introduced. A RAM disk was selected because it does not depend on the availability of a device driver for a secondary storage device. The RAM disk will allow applications to create new files, as well as read and write existing files to a memory disk. Each partition can be configured with any number of RAM disks. Specifically, the LPSK configuration vector describes the number of RAM disks and the access modes (read, write or both) to the disk, for each partition. A limitation of a RAM disk is that it is volatile and will not persist across power cycles. However, it is suitable for a demonstration of file system support for applications.

1. Interfaces

The T-TASI RAM Disk File System will provide the interfaces for file manipulation listed in Table 4 (see Appendix A for the technical specification), which satisfy the requirements for a file storage mechanism identified in Chapter III, Section C. An application in a partition will access the RAM disk storage device via file

management interfaces made available by the T-TASI C Library. This allows the file system to be developed and improved without affecting the applications' design and code.

Table 4. T-TASI RAM Disk File System Interfaces

no.	T-TASI RAM Disk File System	Description
1.	f_open	Open a file for read or write access
2.	f_read	Read data from an opened file
3.	f_write	Write data to an opened file
4.	f_close	Close an opened file
5.	f_unlink	Delete a file from the disk
6.	f_mkdir	Create a directory on the disk
7.	f_rename	Rename a file

2. Dependencies

The T-TASI RAM Disk File System module is implemented in PL2, at a higher privilege level than the application. It depends on *make_ptr*, a Trusted OS Service interface available in PL2, which is used to create a pointer to the memory segment for the RAM disk. The relationships between each of these components are illustrated in Figure 4.

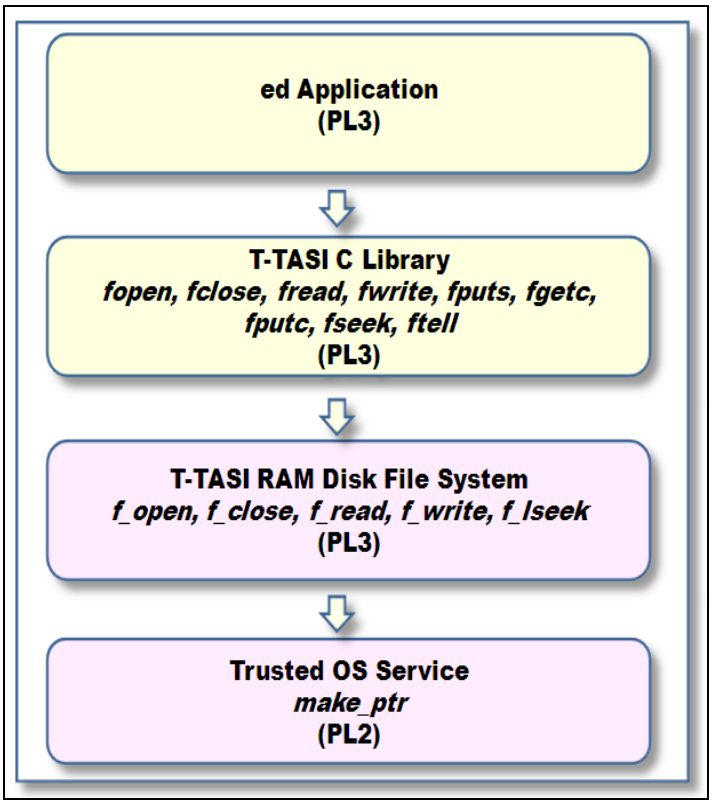

Figure 4. T-TASI RAM Disk File System Relationships

E. T-TASI RAM DISK FILE SYSTEM IMPLEMENTATION

The implementation of the RAM disk file system utilizes the open source project FatFs [25]. The FatFs project was chosen for porting to the T-TASI system because it has a small code base written completely in ANSI C and does not depend on any external libraries. FatFs is a generic File Allocation Table (FAT) file system, developed for small, embedded systems. FatFs supports FAT12, FAT16 and FAT32 file systems. FatFs is free software and is covered by a BSD-style license, which allows use and redistribution with modification [26]. The FatFs license is less restrictive than the "two-clause BSD license," requiring no conditions on its redistribution in binary form. Like the two-clause BSD license, it is a permissive license, which places no restrictions on the use or modification of the source; in particular, unlike the GNU GPL, it places no restrictions on derived works.

1. Initialization

Similar to the T-TASI Application-Level Memory Management component, the offline LPSK Configuration Tool is used to configure the size of the memory segment for the file system module. T-TASI RAM Disk File System module is initialized by the main PL2 routine when the system starts up. During initialization, the memory segment exported by the LPSK is formatted into a FAT16 file system within the file system module.

2. FatFs

FatFs may be viewed as a module, separated into two layers: the file system layer and the disk input/output layer. The file system layer provides application interfaces that manipulate files or directories in a FAT file system. The disk input/output layer provides low-level disk I/O interfaces to the actual storage media. Figure 5 shows the FatFs software architecture (see Appendix A for a detailed interface description and design summary for FatFs).

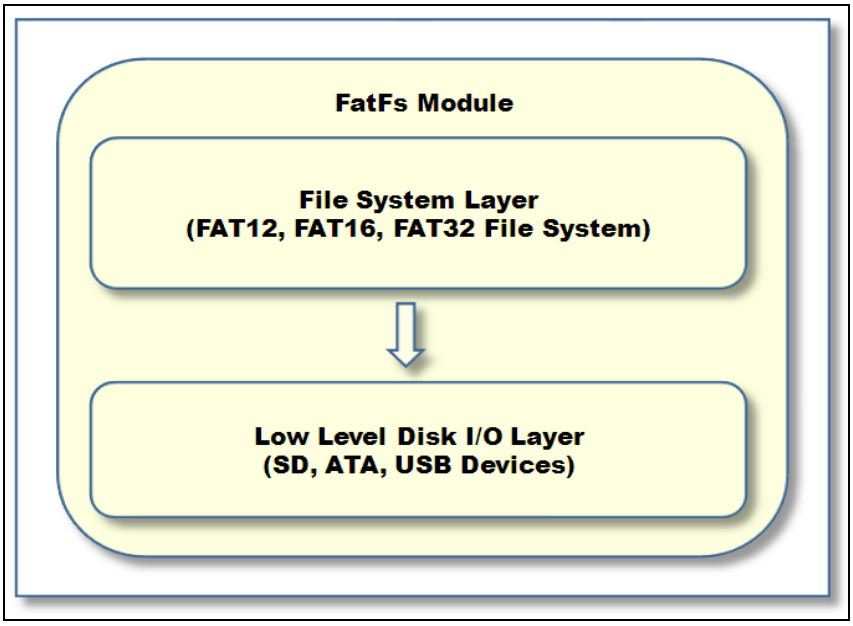

Figure 5. FatFs Software Architecture

3. File System Layer

The file system layer exports the interfaces in Table 4 from PL2 to PL3 via *call gates*, allowing them to be used by the T-TASI C Library. In this layer, a FIL structure is used to represent the internal state of a file. The FIL structure is different from the FILE structure defined by the ISO C standard. The mapping of this FIL structure to the ISO C FILE structure is maintained in the T-TASI C Library, allowing applications to continue to use the standard C FILE structure. Table 5 shows the comparison of FatFs' FIL structure and a standard C FILE structure (see Table 20 for details).

Table 5. FatFs FIL Structure and C Library File Structure

FatFs FIL Structure	**C Library FILE Structure**
typedef struct { FATFS* fs; WORD id; BYTE flag; BYTE pad1; DWORD fptr; DWORD fsize; DWORD org_clust; DWORD curr_clust; DWORD dsect; DWORD dir_sect; BYTE* dir_ptr; BYTE buf[512]; } FIL;	typedef struct { unsigned char* _ptr; int _cnt; unsigned _flag; int _handle; unsigned _bufsize; unsigned short _ungotten; } FILE;

4. Disk I/O Layer

The disk input/output functions in the FatFs module's low-level disk I/O layer, shown in Table 6, were modified to work with an exported memory segment instead of with secondary storage media. This allowed FatFs to be extended, naturally, to serve as a RAM disk file system for the T-TASI system. The disk I/O layer accesses the secondary memory (in this case, a memory segment) in terms of *sectors* [17], each of size 512 bytes. The memory segment used for the file system is divided into blocks of 512 bytes to facilitate this addressing.

Table 6. Disk I/O Functions in FatFs

no.	Interface	Description
1.	disk_initialize	Initialize the disk for use.
2.	disk_read	Read a block of data from the disk.
3.	disk_write	Write a block of data to the disk.

5. Handling Invalid Parameters

There are minimal safety checks in the FatFs module and thus, when illegal parameters or null pointers are passed to its interfaces, the current system halts due to a memory violation error. The code was not modified to be more robust because of time constraints; however, these problems are somewhat mitigated through range validation and other parameter checking performed by the T-TASI C Library.

F. T-TASI C LIBRARY DESIGN

The set of functions included in the initial library design are motivated by our application requirements (see Table 2). When future applications require additional library functions, the library will need to be expanded. ISO C99 standard [24] is the ANSI standard for the C language specification. This specification was used to guide the

design and development of the T-TASI C Library. In particular, all T-TASI C Library interfaces and behaviors will adhere to this standard.

1. Interfaces

The T-TASI C Library will provide the interfaces listed in Table 7, which satisfies the requirements to support *ed* (see Chapter III, Section C). Several other interfaces were implemented in the T-TASI C library even though they are not required by the *ed* application. These functions were implemented because they will be useful for future application development. For example, the function *scanf* is implemented because it is a more general function than *getc* and can be used to implement *getc*, which is required by *ed*.

Each function listed in Table 7 is implemented according to the ISO C99 standard [24]. For each row, the second column provides the C library function interface, the third column gives a brief description of the interface's behavior, and the final column references the relevant section of the ISO C99 standard [24] providing the interface's specification.

Table 7. T-TASI C Library Interfaces

no.	T-TASI C Library Interface	Function Description	Specification
1.	int isspace (int c)	Tests for any character that is a standard white-space character.	[24] §7.4.1.10
2.	int isxdigit (int c)	Tests for any hexadecimal-digit character.	[24] §7.4.1.12
3.	int isdigit (int c)	Tests for any decimal-digit character.	[24] §7.4.1.5
4.	int isalpha (int c)	Tests for any character for which *isupper* or *islower* is true.	[24] §7.4.1.2
5.	int isalnum (int c)	Tests for any character for which *isalpha* or *isdigit* is true.	[24] §7.4.1.1
6.	int islower (int c)	Tests for any character that is a lowercase letter.	[24] §7.4.1.7
7.	int isupper (int c)	Tests for any character that is an uppercase letter.	[24] §7.4.1.11
8.	int tolower (int c)	Converts an uppercase letter to a corresponding lowercase letter.	[24] §7.4.2.1
9.	int atoi (const char* p)	Convert the initial portion of the string pointed to by p to an integer representation.	[24] §7.20.1.2

no.	T-TASI C Library Interface	Function Description	Specification
10.	long int strtol (const char* str, char** endp, int base)	Converts the initial portion of the string pointed to by str to long integer representation.	[24] §7.20.1.4
11.	unsigned long int strtoul (const char* str, char** endp, int base)	Converts the initial portion of the string pointed to by str to unsigned long int representation	[24] §7.20.1.4
12.	char* strcpy (char* dest, char* src)	Copies the string pointed to by src into the array pointed to by dest.	[24] §7.21.2.3

no.	T-TASI C Library Interface	Function Description	Specification
13.	char* strncpy (char* dest, const char* src, unsigned int s)	Copies no more than s characters from the array pointed to by src to the array pointed to by dest.	[24] §7.21.2.4
14.	int strcmp (char* str1, char* str2)	Returns an integer greater than, equal to, or less than 0, if the string pointed to by str1 is greater than, equal to, or less than the string pointed to by str2.	[24] §7.21.4.2
15.	int strncmp (char* str1, char* str2, unsigned int s)	Returns an integer greater than, equal to, or less than 0, if the (possibly null-terminated) string pointed to by str1is greater than, equal to, or less than the (possibly null-terminated) string pointed to by str2.	[24] §7.21.4.4
16.	char* strchr (char* str, int c)	Locates the first occurrence of c in the string pointed to by str.	[24] §7.21.5.2

no.	T-TASI C Library Interface	Function Description	Specification
17.	unsigned int strlen (char* str)	Computes the length of the null-terminated string pointed to by str.	[24] §7.21.6.3
18.	void* malloc (unsigned int s)	Allocates memory space for an object whose size is specified by s.	[24] §7.20.3.3
19.	void free(void* p)	Causes the space pointed to by p to be de-allocated, and made available for further allocation.	[24] §7.20.3.2
20.	void* calloc (unsigned int n, unsigned int s)	Allocates space for an array of n objects, each of whose size is s bytes. The allocated space is initialized to zero.	[24] §7.20.3.1
21.	void* realloc (void* p, unsigned int s)	De-allocates the old object pointed to by p and returns a pointer to a new object that has the size specified by s. The contents of the new object shall be the same as that of the old object prior to de-allocation, up to the lesser of the new and old sizes.	[24] §7.20.3.4

no.	T-TASI C Library Interface	Function Description	Specification
22.	void* memcpy (void* dest, const void* src, unsigned int s)	Copies s bytes from the object pointed to by src into the object pointed to by dest.	[24] §7.21.2.1
23.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	Compares the first s bytes of the object pointed to by ptr1 to the first s bytes of the object pointed to by ptr2.	[24] §7.21.4.1
24.	void* memmove (void* dest, const void* src, unsigned int s)	Copies s bytes from the object pointed to by src into the object pointed to by dest. Copying takes place as if the s bytes from the object pointed to by src are first copied into a temporary array of s bytes that does not overlap the objects pointed to by src and dest, and then the s bytes from the temporary array are copied into the object pointed to by dest.	[24] §7.21.2.2

no.	T-TASI C Library Interface	Function Description	Specification
25.	void* memset (void* ptr, int v, unsigned int s)	Copies the value of v into each of the first s bytes of the object pointed to by ptr.	[24] §7.21.6.1
26.	FILE* fopen (const char* fn, const char* m)	Opens the file whose name is the string pointed to by fn and associates a stream with it.	[24] §7.19.5.3
27.	unsigned int fread (void* ptr, unsigned int s, unsigned int c, FILE* f)	Reads up to c elements from the stream pointed to by f, and writes them into the array pointed to by ptr, where each element is of sizes bytes.	[24] §7.19.8.1

no.	T-TASI C Library Interface	Function Description	Specification
28.	unsigned int fwrite (const void* ptr, unsigned int s, unsigned int c, FILE* f)	Reads up to c elements from the array pointed to by ptr, and writes to the stream pointed to by f, where each element if of size s bytes.	[24] §7.19.8.2
29.	int fseek (FILE* f, long int off, int origin)	Sets the file position indicator for the stream pointed to by f.	[24] §7.19.9.2
30.	int fclose(FILE* f)	Causes the stream pointed to by f to be flushed and the associated file to be closed.	[24] §7.19.5.1
31.	long int ftell(FILE* f)	Obtains the current value of the file position indicator for the stream pointed to by f.	[24] §7.19.9.4

no.	T-TASI C Library Interface	Function Description	Specification
32.	int fputs (const char* s, FILE* f)	Writes the string pointed to by s to the stream pointed to by f.	[24] §7.19.7.4
33.	int fputc (int c, FILE* f)	Writes the character specified by c to the output stream pointed to by f.	[24] §7.19.7.3
34.	int fgetc(FILE* f)	Obtains the next byte (interpreted as an unsigned char converted to an integer) from the stream pointed to by f, and advances the associated file position indicator for the stream.	[24] §7.19.7.1
35.	int printf (const char* format, …)	Writes output to STDOUT, under control of the string pointed to by format that specifies how subsequent arguments are converted for output.	[24] §7.19.6.3

no.	T-TASI C Library Interface	Function Description	Specification
36.	int sprintf (char* b, const char* format, …)	Writes output to the array pointed to by b, under control of the string pointed to by format that specifies how subsequent arguments are converted for output.	[24] §7.19.6.6
37.	int scanf (const char* format, …)	Reads input from STDIN, under control of the string pointed to by format that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input.	[24] §7.19.6.4

2. Dependencies

The T-TASI C library is implemented in PL3 in the T-TASI system; hence, when a function in the library is invoked by an application during runtime, routines in the library will be executed in application process space. Input and output functions such as file or screen manipulation in the library will involve kernel services. Thus, the use of kernel services is abstracted by the library. Figure 6 shows the dependencies of the T-TASI C library on components at each privilege level. The memory management interfaces rely on the T-TASI Application-Level Memory Management module, and the file management interfaces rely on the T-TASI RAM Disk File System. The Application I/O Library already exists as part of the T-TASI system prototype.

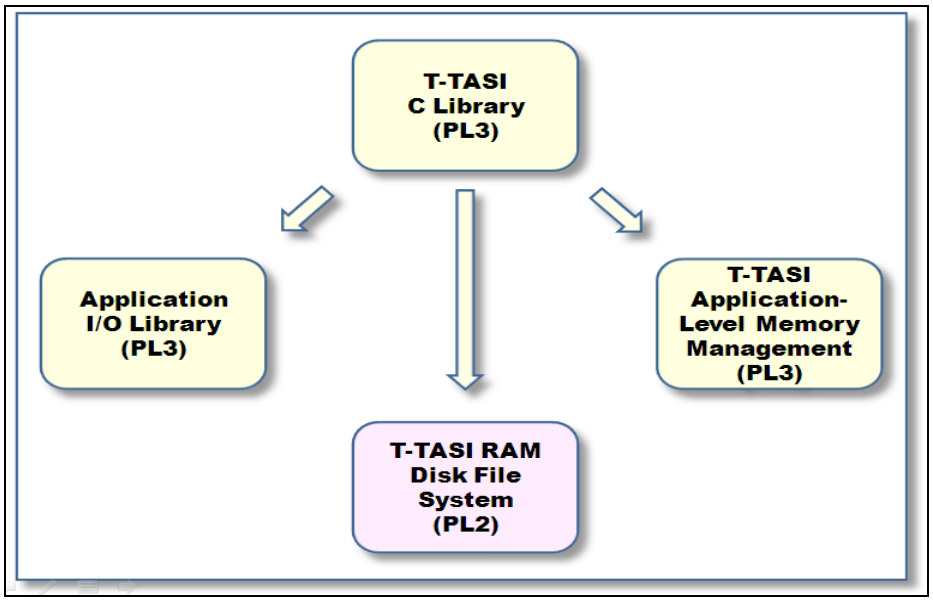

Figure 6. T-TASI C Library Relationships

The specific functions in each of the major services required by the T-TASI C Library are shown in Table 8.

Table 8. External Dependencies of T-TASI C Library

no.	PL	Type of Service	Required Function Interfaces
1.	PL3	Console	tsm_io_getchar, tsm_io_gets, tsm_io_printf (Application I/O Library interfaces)
2.	PL3	Memory Management	get_memory, free_memory (T-TASI Application-Level Memory Management interfaces)
3.	PL2	File Management	f_open, f_read, f_write, f_lseek, f_mount, f_unlink, f_mkdir, f_rename (T-TASI RAM Disk File System interfaces)

G. T-TASI C LIBRARY IMPLEMENTATION

During the development of the T-TASI C Library, open source projects such as the GNU C Library [27], diet libc [28], uClibc [29] and the FreeBSD C Library [30] were inspected to understand how other C libraries are implemented. The source code for regular expression parsing is taken directly from the FreeBSD project and incorporated into the T-TASI C Library. FreeBSD is covered by the permissive, two-clause BSD License [31], which allows the use, modification and redistribution of binary and source.

The functions required by the *ed* application are classified as utility, file, memory, console, process, and signal as shown in Table 2. Utility functions have no dependency on other components and are straightforward routines implemented directly in the T-TASI C Library. The following sub-sections describe how the other functions are implemented.

1. File Functions

Applications manipulate files in the file system through functions provided by the T-TASI C Library. These interfaces, in turn, invoke the relevant PL2 functions exported

by the T-TASI RAM Disk File System. Table 9 provides the mapping of the T-TASI C library file functions to the PL2 interfaces for T-TASI RAM Disk File System functions.

Table 9. Mapping between the C Library Functions and the File System Functions

no.	**T-TASI C Library File Function (PL3)**	**T-TASI RAM Disk File System Functions (PL2)**
1.	*fopen*	*f_open*
2.	*fclose*	*f_close*
3.	*fread*	*f_read*
4.	*fgetc*	*f_read*
5.	*fwrite*	*f_write*
6.	*fputs*	*f_write*
7.	*fputc*	*f_write*
8.	*fseek*	*f_lseek*

2. Memory Functions

Applications access memory services through those T-TASI C Library interfaces related to memory. The T-TASI C Library memory functions *malloc* and *free* are supported by the T-TASI Application-Level Memory Management interfaces, *get_memory* and *free_memory* respectively. When an application invokes *malloc* (or *free*), the interface *get_memory* (or *free_memory*) in the T-TASI Application-Level Memory Management module will be invoked.

3. Console Functions

The T-TASI C Library console functions *scanf*, *printf*, and *puts* are supported by the T-TASI Application I/O Library. Specifically, the input function, *scanf*, is supported by *tsm_io_getchar* and the output functions *printf* and *puts* are supported by *tsm_io_printf*.

4. Signal and Process Functions

Due to the characteristics and limitations of the current T-TASI system prototype, not all of the functions required to support *ed* have be implemented. Those functions that cannot be implemented have been replaced with empty stub functions or, to prevent errors during the linking process, have been implemented by functions that simply return a default value. Signal and process functions dealing with pipes, signals, jump and the shell environment have been stubbed in this fashion. Table 10 shows the list of functions required by *ed* that are not implemented in the T-TASI C Library. Through testing, it has been determined that most functionality of *ed* has not been affected by these implementation decisions. The rationale for postponing the implementation of each function and the effects of their new behavior are summarized in Table 10. For each row, the second column provides the interface for the unsupported function, the third column provides a brief description of the interface's behavior, the fourth column describes the type of function implementing the interface, the fifth column provides the rationale for not implementing the standard function behavior, and the final column describes the behavior of the replacement function.

Table 10. T-TASI C Library Stubbed Functions and Their Effects on the *ed* Application

no.	**C Library Interface**	**Interface Description**	**Replaced by**	**Reasons**	**Behavior of implemented function**
1.	*setjmp*	Save the current application environment for *longjmp*	Empty stub function	Requires modifying registers such as ESP and EIP, which is currently not allowed by the kernel.	The capability of *ed* to gracefully shutdown when the system generates a hang up signal is disabled.
2.	*longjmp*	Restores the application environment set by *setjmp*	Empty stub function	Requires modifying registers such as ESP and EIP, which is currently not allowed by the kernel.	The capability of *ed* to gracefully shutdown when the system generates a hang up signal is disabled.
3.	*exit*	Terminate the existing application	Empty stub function	In the current T-TASI system prototype, an application in a partition cannot be terminated.	*ed* will stay resident in the partition.

no.	C Library Interface	Interface Description	Replaced by	Reasons	Behavior of implemented function
4.	*getenv*	Get the application environment of a given value	Return an empty string to the application.	No implementation of an application environment is available in the current T-TASI system prototype.	Used to locate the home directory of the user in order to save opened file when the system generates a hang up signal.
5.	*isatty*	Test whether a given device is a terminal	Always return true.	Not a standard C library function.	The capability of *ed* to gracefully shutdown when the system generates a hang up signal is disabled.
6.	*pathconf*	Get the path name configuration and limits	Always return 256.	No implementation of an application environment is available in the current T-TASI system prototype.	The size of path name is hard-coded to 256 when this function is invoked. 256 bytes is the maximum path of the implemented file system.

no.	C Library Interface	Interface Description	Replaced by	Reasons	Behavior of implemented function
7.	*pclose*	Close a pipe stream	Empty stub function	No implementation of pipes is available in the current T-TASI system prototype.	No data can be piped to the *ed* application from a shell.
8.	*popen*	Open a pipe stream	Empty stub function	No implementation of pipes is available in the current T-TASI system prototype.	No data can be piped to the *ed* application from a shell.
9.	*setvbuf*	Assign buffering to a stream	Always return success	Used by the *ed* application to avoid contention when using pipes.	No data can be piped to the *ed* application from a shell.
10.	*sigaddset*	Add a signal to a signal set	Function invocation is commented out	Used by the *ed* application to register and handle hang up signals from the system.	The capability of *ed* to gracefully shutdown when the system generates a hang up signal is disabled.
11.	*sigemptyset*	Remove a signal from a signal set	Function invocation is commented out		

no.	C Library Interface	Interface Description	Replaced by	Reasons	Behavior of implemented function
12.	*sigaction*	Specify an action to be associated with a given signal	Function invocation is commented out		
13.	*sigprocmask*	Change a blocked signal	Function invocation is commented out		
14.	*ioctl*	Control a stream device	Empty stub function	Not a standard C library function.	The capability of *ed* to gracefully shutdown when the system generates a hang up signal is disabled.
15.	*system*	Issue an external command from the application	Empty stub function	No implementation of environment of a shell in a partition in the current T-TASI system prototype.	The *ed* application cannot issue and execute commands to a shell.

no.	C Library Interface	Interface Description	Replaced by	Reasons	Behavior of implemented function
16.	*setlocale*	Set the application specific locale	Empty stub function	No implementation of locale in the current T-TASI system prototype.	A default international environment is used by *ed* application.

5. Handling Invalid Parameters

For most of the functions in the standard C library that have pointer inputs, their behaviors on null inputs are unspecified. For most UNIX systems, when a null pointer is passed as a parameter to a function, a segmentation fault occurs when the function dereferences that pointer. In response, the kernel generates a signal to the process notifying it of the memory violation. By default, the process dumps its working memory to a file and terminates. In the current T-TASI system prototype, when a process has a memory access violation, an interrupt will be generated (interrupt number 13) and the system will halt. We note that this is merely the behavior of the current prototype and, in the future, the T-TASI system will terminate the offending process and the system will not halt. Additional safety checks for null pointers are implemented for these functions in the current T-TASI C Library. Specifically, when a null pointer would cause a segmentation fault, additional checks will return a failure code back to the application instead of referencing the pointer, thereby preventing the system from halting.

In the case of invalid or bad pointers, no additional checks are provided. The current T-TASI system will halt when an invalid or bad pointer is encountered in these functions. Additional checks are not implemented because at the PL3 level the library is not able to discern whether a particular memory access is permissible or not.

H. SUMMARY

This chapter presented the design of the three main software libraries used to port *ed* to run on the T-TASI system, and a description of how each is implemented. These three software libraries were developed utilizing a combination of newly developed and pre-existing modules. In particular, the FatFs project was used to implement the T-TASI RAM Disk File System, and code for regular parsing from FreeBSD was used to implement some interfaces of the T-TASI C Library. The next chapter describes test plans and testing results for each software component.

THIS PAGE INTENTIONALLY LEFT BLANK

V. TESTING

This chapter describes the test design and the outcome of testing for the components developed in this project. A description of how to perform each set of tests is provided in Appendix C.

A. TESTING APPROACH

This section describes the tests designed for the software artifacts of this project. Tests are separated into two major types. First, component tests are those designed to verify each component's compliance with its specification. Integration tests, in which all components are combined and tested together, provide evidence of the functional correctness of the whole system.

The components to be tested are:

1. T-TASI C Library
2. T-TASI Application-Level Memory Management
3. T-TASI RAM Disk File System
4. *ed* Application

The component tests have been implemented to run as automated unit tests. A unit testing methodology provides evidence that the source code is working correctly by dividing the source into its smallest testable parts (or units). Each test is classified as either a Functional (F) or Exception (E) test. Functional tests are designed to verify the interface's correctness on valid inputs and exception tests are designed for negative cases and behavior on invalid inputs.

B. TESTING LIMITATIONS

Function behavior outside the interface's specification are, for the most part, not tested. For example, for most functions, the function's behavior when manipulating a bad pointer is unspecified. For the T-TASI system, the general behavior is known—an

interrupt is generated when the application tries to access memory that does not belong to its partition—but verifying this behavior is not part of the interface test plan. As any behavior would satisfy this underspecified condition, such a test is always, trivially, passed. As another example, for the *strcpy* interface, when the size of the destination buffer is smaller than the source being copied, this will result in a memory overflow with unpredictable effects. If the memory overflows to the application stack memory, corruption of application memory occurs. If the memory overflows to a memory region not belonging to the partition, then the CPU will generate an interrupt.

Boundary testing, which is part of exception testing, for memory limits is not conducted for all tests due to the memory constraints of the development machine, which is only equipped with 4 gigabytes of memory. In particular, to test the *memcpy* interface using the largest possible value for the buffer size parameter requires a system with at least 8 gigabytes of memory for its source and destination buffers. In general, when the behavior of a function on certain inputs is unspecified or the ultimate effects of the function on those inputs are variable, these exception tests are not part of the interface test plans.

C. T-TASI C LIBRARY TEST PLAN

The objectives of the following tests are to verify that the implementation of the T-TASI C Library interfaces conform to their specification.

Table 11 provides a summary of the test cases conducted for the T-TASI C Library. Each row provides a description of the test case performed: the first column refers to the test case number, the second column refers to the T-TASI C library interface tested, the third column refers to the type of test conducted, the fourth column provides a description of the parameters used in test, the fifth column gives the expected test result, and the final column provides the action result of the test. Subsequent tables in this chapter use the same column headings and have the same meaning as described here.

Table 11. T-TASI C Library Test Cases

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
1.	int isspace (int c)	F	Parameter c is a space	Return 1	Passed
2.	int isspace (int c)	F	Parameter c is value ‘a’, i.e., not a space	Return 0	Passed
3.	int isxdigit (int c)	F	Parameter c is a hexadecimal digit, 0x0a	Return 1	Passed
4.	int isxdigit (int c)	F	Parameter c is not a hexadecimal digit, ‘k’	Return 0	Passed
5.	int isdigit (int c)	F	Parameter c is a digit, 7	Return 1	Passed
6.	int isdigit (int c)	F	Parameter c is not a digit, ‘a’	Return 0	Passed
7.	int isalpha (int c)	F	Parameter c is a letter, ‘a’	Return 1	Passed
8.	int isalpha (int c)	F	Parameter c is not a letter, 7	Return 0	Passed
9.	int isalnum (int c)	F	Parameter c is a letter, ‘a’	Return 1	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
10.	int isalnum (int c)	F	Parameter c is a digit, 7	Return 1	Passed
11.	int isalnum (int c)	F	Parameter c is a space, ‘ ’	Return 0	Passed
12.	int islower (int c)	F	Parameter c is a lower case letter, ‘a’	Return 1	Passed
13.	int islower (int c)	F	Parameter c is an upper case letter, ‘A’	Return 0	Passed
14.	int islower (int c)	E	Parameter c is not a letter, ‘?’	Return 0	Passed
15.	int isupper (int c)	F	Parameter c is an upper case letter, ‘A’	Return 1	Passed
16.	int isupper (int c)	F	Parameter c is a lower case letter, ‘a’	Return 0	Passed
17.	int isupper (int c)	E	Parameter c is not a letter, ‘?’	Return 0	Passed
18.	int tolower (int c)	F	Parameter c is an upper case letter, ‘A’	Return 97, ASCII value of character ‘a’	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
19.	int tolower (int c)	E	Parameter c is a lower case letter, ‘b’	Return 98, ASCII value of character ‘b’	Passed
20.	int tolower (int c)	E	Parameter c is a digit character, ‘7’	Return 55, ASCII value of character ‘7’	Passed
21.	int atoi (const char* p)	F	Parameter p is a pointer to a string value, “12345”	Return integer value 12345	Passed
22.	int atoi (const char* p)	F	Parameter p is a pointer to a string value, “-0”	Return integer value 0	Passed
23.	int atoi (const char* p)	F	Parameter p is a pointer to a string value, “2147483647”	Return integer value 2147483647	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
24.	int atoi (const char* p)	F	Parameter p is a pointer to a string value, “-2147483648”	Return integer value -2147483648	Passed
25.	int atoi (const char* p)	E	Parameter p is a pointer to a string value, “aabbcc”	Return 0	Passed
26.	int atoi (const char* p)	E	Parameter p is a pointer to a string value, “9999999999”	Return 2147483647 (maximum integer value)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
27.	int atoi (const char* p)	E	Parameter p is a pointer to a string value, "-9999999999"	Return -2147483648 (minimum integer value)	Passed
28.	int atoi (const char* p)	E	Parameter p is a null pointer	Return 0	Passed
29.	long int strtol (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value "12345", parameter endp is a character pointer, parameter base is 10	Return 12345	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
30.	long int strtol (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “12345”, parameter endp is a character pointer, parameter base is 16	Return 74565 (12345 interpreted in base 16)	Passed
31.	long int strtol (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “12345”, parameter endp is a character pointer, parameter base is 8	Return 5349 (12345 interpreted in base 8)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
32.	long int strtol (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “2000000000”, parameter endp is a character pointer, parameter base is 10	Return 2000000000	Passed
33.	long int strtol (const char* str, char** endp, int base)	E	Parameter str is a pointer to a string value “2000000000”, parameter endp is a character pointer, parameter base is 16	Return 2147483647 (maximum long value)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
34.	long int strtol (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “2000000000”, parameter endp is a character pointer, parameter base is 8	Return 268435456 (2000000000 interpreted in base 8)	Passed
35.	long int strtol (const char* str, char** endp, int base)	E	Parameter str is a pointer to a null pointer, parameter endp is a character pointer, parameter base is 10	Return 0	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
36.	long int strtol (const char* str, char** endp, int base)	E	Parameter str is a pointer to a string value “9999999999”, parameter endp is a character pointer, parameter base is 10	Return 2147483647 (maximum long value)	Passed
37.	long int strtol (const char* str, char** endp, int base)	E	Parameter str is a pointer to a string value “-9999999999”, parameter endp is a character pointer, parameter base is 10	Return -2147483648 (minimum long value)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
38.	unsigned long int strtoul (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “12345”, parameter endp is a character pointer, parameter base is 10	Return 12345	Passed
39.	unsigned long int strtoul (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “12345”, parameter endp is a character pointer, parameter base is 16	Return 74565 (12345 interpreted in base 16)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
40.	unsigned long int strtoul (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “12345”, parameter endp is a character pointer, parameter base is 8	Return 5349 (12345 interpreted in base 8)	Passed
41.	unsigned long int strtoul (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “0”, parameter endp is a character pointer, parameter base is 10	Return 0	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
42.	unsigned long int strtoul (const char* str, char** endp, int base)	F	Parameter str is a pointer to a string value “4294967295”, parameter endp is a character pointer, parameter base is 10	Return 4294967295	Passed
43.	unsigned long int strtoul (const char* str, char** endp, int base)	E	Parameter str is a pointer to a null pointer, parameter endp is a character pointer, parameter base is 10	Return 0	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
44.	unsigned long int strtoul (const char* str, char** endp, int base)	E	Parameter str is a pointer to a string value “9999999999”, parameter endp is a character pointer, parameter base is 10	Return 4294967295 (maximum value)	Passed
45.	unsigned long int strtoul (const char* str, char** endp, int base)	E	Parameter str is a pointer to a string value “-9999999999”, parameter endp is a character pointer, parameter base is 10	Return 4294967295 (maximum value)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
46.	char* strcpy (char* dest, char* src)	F	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a pointer to the string value of “Hello World”	Return pointer to “Hello World”, this pointer is the same as dest	Passed
47.	char* strcpy (char* dest, char* src)	F	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a pointer to the string value of “Hello”	Return pointer to “Hello”, this pointer is the same as dest	Passed
48.	char* strcpy (char* dest, char* src)	E	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a null pointer.	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
49.	char* strcpy (char* dest, char* src)	E	Parameter dest is a null pointer. Parameter src is a pointer to the string value of “Hello World”	Return null value	Passed
50.	char* strcpy (char* dest, char* src)	E	Parameter dest is a null pointer. Parameter src is a null pointer	Return null value	Passed
51.	char* strncpy (char* dest, const char* src, unsigned int s)	F	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a pointer to the string value of “Hello World”. Parameter s is of value 12	Return pointer to “Hello World”, this pointer is the same as dest	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
52.	char* strncpy (char* dest, const char* src, unsigned int s)	F	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a pointer to the string value of “Hello World”. Parameter s is of value 11	Return pointer to “Hello World”, this pointer is the same as dest	Passed
53.	char* strncpy (char* dest, const char* src, unsigned int s)	F	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a pointer to the string value of “Hello World”. Parameter s is of value 4	Return pointer to “Hell”, this pointer is the same as dest	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
54.	char* strncpy (char* dest, const char* src, unsigned int s)	E	Parameter dest is a character array of size 12 initialized to value 0. Parameter src is a null pointer. Parameter s is of value 12	Return null value	Passed
55.	char* strncpy (char* dest, const char* src, unsigned int s)	E	Parameter dest is a null pointer. Parameter src is a pointer to the string value of “Hello World”. Parameter s is of value 12	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
56.	char* strncpy (char* dest, const char* src, unsigned int s)	E	Parameter dest is a null pointer. Parameter src is a null pointer. Parameter s is of value 12	Return null value	Passed
57.	int strcmp (char* str1, char* str2)	F	Parameter str1 is a string value of “Hello”. Parameter str2 is a string value of “Hello”	Return 0	Passed
58.	int strcmp (char* str1, char* str2)	F	Parameter str1 is a string value of “Hello”. Parameter str2 is a string value of “World”	Return -15 (the numeric difference of ‘H’ and ‘W’)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
59.	int strcmp (char* str1, char* str2)	F	Parameter str1 is a string value of “World”. Parameter str2 is a string value of “Hello”	Return 15 (the numeric difference of ‘W’ and ‘H’)	Passed
60.	int strcmp (char* str1, char* str2)	E	Parameter str1 is a string value of “Hello”. Parameter str2 is a null pointer	Return -1 (parameter cannot be null)	Passed
61.	int strcmp (char* str1, char* str2)	E	Parameter str1 is a null pointer. Parameter str2 is a string value of “World”. Parameter str2 is a null pointer	Return -1 (parameter cannot be null)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
62.	int strcmp (char* str1, char* str2)	E	Parameter str1 is a null pointer. Parameter str2 is a null pointer	Return -1 (parameter cannot be null)	Passed
63.	int strncmp (char* str1, char* str2, unsigned int s)	F	Parameter str1 is a string value of “Hello”. Parameter str2 is a string value of “Hello”. Parameter s is of value 5	Return 0	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
64.	int strncmp (char* str1, char* str2, unsigned int s)	F	Parameter str1 is a string value of “Hello”. Parameter str2 is a string value of “Hella”. Parameter s is of value 5	Return 14 (the numeric difference of ‘o’ and ‘a’)	Passed
65.	int strncmp (char* str1, char* str2, unsigned int s)	F	Parameter str1 is a string value of “Hella”. Parameter str2 is a string value of “Hello”. Parameter s is of value 5	Return -14 (the numeric difference of ‘a’ and ‘o’)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
66.	int strncmp (char* str1, char* str2, unsigned int s)	E	Parameter str1 is a string value of “Hello”. Parameter str2 is a null pointer”. Parameter s is of value 5	Return -1 (parameter cannot be null)	Passed
67.	int strncmp (char* str1, char* str2, unsigned int s)	E	Parameter str1 is a null pointer. Parameter str2 is a string value of “Hello”. Parameter str2 is a null pointer. Parameter s is of value 5	Return -1 (parameter cannot be null)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
68.	int strncmp (char* str1, char* str2, unsigned int s)	E	Parameter str1 is a null pointer. Parameter str2 is a null pointer. Parameter s is of value 5	Return -1	Passed
69.	char* strchr (char* str, int c)	F	Parameter str is a string value of “Hello World”. Parameter c is a letter “e”	Return the string value of “ello”, the pointer returned is the position of the letter ‘e’ in the original str	Passed
70.	char* strchr (char* str, int c)	F	Parameter str is a string value of “Hello World”. Parameter c is a letter ‘K’.	Return null value (‘K’ not found in str)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
71.	char* strchr (char* str, int c)	E	Parameter str is a null pointer. Parameter c is letter ‘W’.	Return null value	Passed
72.	unsigned int strlen (char* str)	F	Parameter str is a string value of “Hello”	Return 5	Passed
73.	unsigned int strlen (char* str)	E	Parameter str is a null pointer	Return 0	Passed
74.	void* malloc (unsigned int s)	F	Parameter s is of value 1024	Return a pointer to a memory region of size 1024	Passed
75.	void* malloc (unsigned int s)	E	Parameter s is of value 0	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
76.	void free(void* p)	F	Parameter p is a pointer previously malloc with 1024 bytes	No return value	Passed
77.	void free(void* p)	E	Parameter p is a null pointer	No return value	Passed
78.	void* calloc (unsigned int n, unsigned int s)	F	Parameter n is a digit of value 10. Parameter s is a digit of value 10	Return a pointer to a memory region of size 100 (10 x 10). The memory region is initialized to value 0.	Passed
79.	void* calloc (unsigned int n, unsigned int s)	E	Parameter n is a digit of value 0. Parameter s is a digit of value 10	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
80.	void* calloc (unsigned int n, unsigned int s)	E	Parameter n is a digit of value 10. Parameter s is a digit of value 0	Return null value	Passed
81.	void* calloc (unsigned int n, unsigned int s)	E	Parameter n is a digit of value 0. Parameter s is a digit of value 0	Return null value	Passed
82.	void* realloc (void* p, unsigned int s)	F	Parameter p is a null pointer. Parameter s is a digit of value 1024	Return a pointer to a memory region of size 1024	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
83.	void* realloc (void* p, unsigned int s)	F	Parameter p is a pointer to a previously malloced memory region of size 1024. Parameter s is a digit of value 2048	Return a pointer to a memory region of size 2048	Passed
84.	void* realloc (void* p, unsigned int s)	F	Parameter p is a pointer to a previously malloced memory region of size 1024. Parameter s is a digit of value 0	Return a null value	Passed
85.	void* realloc (void* p, unsigned int s)	F	Parameter p is a pointer to a previously malloced memory region of size 1024. Parameter s is a digit of value 512	Return a pointer to a memory region of size 512	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
86.	void* realloc (void* p, unsigned int s)	E	Parameter p is a null pointer. Parameter s is a digit of value 0	Return a null value	Passed
87.	void* memcpy (void* dest, const void* src, unsigned int s)	F	Parameter dest is a pointer to a memory region of size 16. Parameter src is a pointer to a string value of “Hello World”. Parameter s is a digit of value 12	Return dest pointer containing “Hello World”	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
88.	void* memcpy (void* dest, const void* src, unsigned int s)	F	Parameter dest is a pointer to a memory region of size 12. Parameter src is a pointer to a string value of “Hello World”. Parameter s is a digit of value 12	Return a pointer to the memory region containing “Hello World”	Passed
89.	void* memcpy (void* dest, const void* src, unsigned int s)	E	Parameter dest is a pointer to a memory region of size 12. Parameter src is a null pointer. Parameter s is a digit of value 12	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
90.	void* memcpy (void* dest, const void* src, unsigned int s)	E	Parameter dest is a null pointer. Parameter src is a pointer to a string value of “Hello World”. Parameter s is a digit of value 12	Return null value	Passed
91.	void* memcpy (void* dest, const void* src, unsigned int s)	E	Parameter dest is a null pointer. Parameter src is a null pointer. Parameter s is a digit of value 12	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
92.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	F	Parameter ptr1 is a string value of “Hello”. Parameter str2 is a string value of “Hello”. Parameter s is of value 5	Return 0	Passed
93.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	F	Parameter ptr1 is a string value of “Hello”. Parameter ptr2 is a string value of “Hella”. Parameter s is of value 5	Return 14	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
94.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	E	Parameter ptr1 is a string value of “Hella”. Parameter ptr2 is a string value of “Hello”. Parameter s is of value 5	Return -14	Passed
95.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	E	Parameter ptr1 is a string value of “Hello”. Parameter ptr2 is a null pointer”. Parameter s is of value 5	Return -1	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
96.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	E	Parameter ptr1 is a null pointer. Parameter str2 is a string value of “Hello”. Parameter str2 is a null pointer. Parameter s is of value 5	Return -1	Passed
97.	int memcmp (const void* ptr1, const void* ptr2, unsigned int s)	E	Parameter ptr1 is a null pointer. Parameter str2 is a null pointer. Parameter s is of value 5	Return -1	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
98.	void* memmove (void* dest, const void* src, unsigned int s)	F	Parameter dest is a pointer to a memory region of size 12. Parameter src is a pointer to a string value of “Hello World”. Parameter s is a digit of value 12	Return a pointer to the memory region containing “Hello World”	Passed
99.	void* memmove (void* dest, const void* src, unsigned int s)	E	Parameter dest is a pointer to a memory region of size 12. Parameter src is a null pointer. Parameter s is a digit of value 12	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
100.	void* memmove (void* dest, const void* src, unsigned int s)	E	Parameter dest is a null pointer. Parameter src is a pointer to a string value of “Hello World”. Parameter s is a digit of value 12	Return null value	Passed
101.	void* memset (void* ptr, int v, unsigned int s)	F	Parameter ptr is a pointer to a memory region of size 10. Parameter v is a character of value ‘a’. Parameter s is a digit of value 10	Return ptr pointing to the original memory region initialized to ‘a’	Passed
102.	void* memset (void* ptr, int v, unsigned int s)	E	Parameter ptr is null pointer. Parameter v is a character of value ‘a’. Parameter s is a digit of value 10	Return null value	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
103.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “file.txt”. The file named “file.txt” exists in the disk. Parameter m is a pointer to a string value “r”	Return a FILE pointer	Passed
104.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “r”	Return null value (Cannot open a non existing file for reading)	Passed
105.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “r+”	Return null value (Cannot open a non existing file for reading / appending)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
106.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “w”	Return a FILE pointer	Passed
107.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “w+”	Return a FILE pointer	Passed
108.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “a”	Return a FILE pointer	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
109.	FILE* fopen (const char* fn, const char* m)	F	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “a+”	Return a FILE pointer	Passed
110.	FILE* fopen (const char* fn, const char* m)	E	Parameter fn is a null pointer. Parameter m is a null pointer	Return a null value (invalid parameters)	Passed
111.	FILE* fopen (const char* fn, const char* m)	E	Parameter fn is a pointer to a string value “nofile.txt”. The file named “nofile.txt” does not exist in the disk. Parameter m is a pointer to a string value “zzz”	Return a null value (invalid file open mode)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
112.	unsigned int fread (void* ptr, unsigned int s, unsigned int c, FILE* f)	F	Parameter ptr is a pointer to a memory region of size 10. Parameter s is digit of a value 1. Parameter c is a digit of value 10. Parameter f is a FILE pointer previously returned by fopen	Return 10 (the size of bytes read from the file)	Passed
113.	unsigned int fread (void* ptr, unsigned int s, unsigned int c, FILE* f)	E	Parameter ptr is a null pointer. Parameter s is digit of value 1. Parameter c is a digit of value 10. Parameter f is a FILE pointer previously opened by fopen	Return 0 (invalid parameters)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
114.	unsigned int fread (void* ptr, unsigned int s, unsigned int c, FILE* f)	E	Parameter ptr is a null pointer. Parameter s is digit of a value 1. Parameter c is a digit of value 10. Parameter f is a null pointer	Return 0 (invalid parameters)	Passed
115.	unsigned int fwrite (const void* ptr, unsigned int s, unsigned int c, FILE* f)	F	Parameter ptr is a pointer to a memory region of size 10 initialized to ‘a’. Parameter s is digit of a value 1. Parameter c is a digit of value 10. Parameter f is a FILE pointer previously opened by fopen	Return 10 (the number of bytes written to file)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
116.					
117.	unsigned int fwrite (const void* ptr, unsigned int s, unsigned int c, FILE* f)	E	Parameter ptr is a null pointer. Parameter s is digit of a value 1. Parameter c is a digit of value 10. Parameter f is a FILE pointer previously opened by fopen	Return 0 (invalid parameters)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
118.	unsigned int fwrite (const void* ptr, unsigned int s, unsigned int c, FILE* f)	E	Parameter ptr is a null pointer. Parameter s is digit of a value 1. Parameter c is a digit of value 10. Parameter f is a null pointer	Return 0 (invalid parameters)	Passed
119.	int fseek (FILE* f, long int off, int origin)	F	Parameter is a FILE pointer previously returned by fopen. Parameter off is a digit of value 1. Parameter origin is a digit of value SEEK_SET	Return 0 (success)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
120.	int fseek (FILE* f, long int off, int origin)	F	Parameter is a FILE pointer previously returned by fopen. Parameter off is a digit of value 1. Parameter origin is a digit of value SEEK_CUR	Return 0 (success)	Passed
121.	int fseek (FILE* f, long int off, int origin)	F	Parameter is a FILE pointer previously returned by fopen. Parameter off is a digit of value 1. Parameter origin is a digit of value SEEK_END	Return 0 (success)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
122.	int fseek (FILE* f, long int off, int origin)	E	Parameter is a null pointer. Parameter off is a digit of value 1. Parameter origin is a digit of value SEEK_END	Return -1 (failure)	Passed
123.	int fseek (FILE* f, long int off, int origin)	E	Parameter is a FILE pointer previously returned by fopen. Parameter off is a digit of value 1. Parameter origin is a digit of value 9999	Return -1 (failure)	Passed
124.	int fclose(FILE* f)	F	Parameter f is a FILE pointer previously returned by fopen	Return 0 (success)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
125.	int fclose(FILE* f)	E	Parameter f is a null pointer	Return -1 (failure)	Passed
126.	long int ftell(FILE* f)	F	Parameter f is a FILE pointer previously returned by fopen	Return 0 (current file pointer position)	Passed
127.	long int ftell(FILE* f)	E	Parameter f is a null pointer	Return -1 (invalid parameter)	Passed
128.	int fputs (const char* s, FILE* f)	F	Parameter s is a pointer to a string value “Hello World”. Parameter f is a FILE pointer previously returned by fopen	Return 11 (the number of bytes written to the file)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
129.	int fputs (const char* s, FILE* f)	E	Parameter s is a null pointer. Parameter f is a FILE pointer previously returned by fopen	Return -1 (invalid parameter)	Passed
130.	int fputs (const char* s, FILE* f)	E	Parameter s is a null pointer. Parameter f is a null pointer	Return -1 (invalid parameters)	Passed
131.	int fputc (int c, FILE* f)	F	Parameter c is a character of value ‘a’. Parameter f is a FILE pointer previously returned by fopen	Return 97 (ASCII value of letter ‘a’)	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
132.	int fputc (int c, FILE* f)	E	Parameter c is a character of value ‘a’. Parameter f is a null pointer	Return -1 (invalid parameter)	Passed
133.	int fgetc(FILE* f)	F	Parameter f is a FILE pointer previously returned by fopen	Return a non -1 value	Passed
134.	int fgetc(FILE* f)	E	Parameter f is a null pointer	Return -1	Passed
135.	int printf (const char* format, …)	F	Parameter format is a string value “%s”. The last parameter is a string value “Hello World”	Return 11 and prints “Hello World” to the screen	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
136.	int printf (const char* format, …)	F	Parameter format is a string value “%d”. The last parameter is a digit of value 1234	Return 4 and prints “1234” to the screen	Passed
137.	int printf (const char* format, …)	F	Parameter format is a string value “%c”. The last parameter is a character of value ‘z’	Return 1 and prints “z” to the screen	Passed
138.	int sprintf (char* b, const char* format, …)	F	Parameter b is a pointer to a memory region of size 20. Parameter format is a string value “%s”. The last parameter is a string value “Hello World”	Return 11	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
139.	int sprintf (char* b, const char* format, …)	F	Parameter b is a pointer to a memory region of size 20. Parameter format is a string value “%d”. The last parameter is a digit of value 1234	Return 4	Passed
140.	int sprintf (char* b, const char* format, …)	F	Parameter b is a pointer to a memory region of size 20. Parameter format is a string value “%c”. The last parameter is a character of value ‘z’	Return 1	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
141.	int sprintf (char* b, const char* format, …)	E	Parameter b is a null pointer. Parameter format is a string value “%c”. The last parameter is a character of value ‘z’	Return -1	Passed
142.	int scanf (const char* format, …)	F	Parameter format is a string value “%s”. The last parameter is a pointer to a memory region of size 1024	Return 3 if “abc” is entered on the screen	Passed
143.	int scanf (const char* format, …)	F	Parameter format is a string value “%d”. The last parameter is a pointer to the address of an integer	Return 2 if “12” is entered on the screen	Passed

Test no.	T-TASI C Library Interface	Type	Description	Expected Result	Test Result
144.	int scanf (const char* format, …)	F	Parameter format is a string value “%c”. The last parameter is a pointer to the address of an character	Return 1 if “c” is entered on the screen	Passed

D. T-TASI APPLICATION-LEVEL MEMORY MANAGEMENT TEST PLAN

The objective of the following test plan is to verify that the implemented T-TASI Application-Level Memory Management interfaces conform to their specification. Table 12 provides a summary of the different tests that were conducted for the T-TASI Application-Level Memory Management.

Table 12. T-TASI Application-Level Memory Management Test Cases

Test no.	Interface	Type	Description	Expected Result	Test Result
145.	int get_memory (unsigned int s, void** ptr)	F	Parameter s is of value 1024	Return 1 and the ptr will point to a memory region of 1024 bytes	Passed
146.	int get_memory (unsigned int s, void** ptr)	E	Parameter s is of value 0	Return 1	Passed
147.	int get_memory (unsigned int s, void** ptr)	E	Parameter s is of value 4294967296 (maximum unsigned integer value)	Return 0 (Not enough memory for allocation)	Passed

Test no.	Interface	Type	Description	Expected Result	Test Result
148.	int free_memory (void* p)	F	Parameter p is a pointer to a memory of size 1024 bytes previously returned by get_memory	Return 0. Memory is freed and total memory increased by 1029 bytes	Passed
149.	int free_memory (void* p)	E	Parameter p is a null pointer.	Return 1	Passed

E. T-TASI RAM DISK FILE SYSTEM TEST PLAN

The objectives of the following tests are to verify that the implemented T-TASI RAM Disk File System interfaces conform to their requirements. Table 13 provides a summary of the different tests that were conducted for the T-TASI RAM Disk File System.

Table 13. T-TASI RAM Disk File System Test Cases

Test no.	Interface	Type	Description	Expected Result	Test Result
150.	int f_mount (unsigned char b, FATFS* fs)	F	Parameter b is value 0. Parameter fs is pointer to a FATFS structure	RAM drive is mounted	Passed
151.	int f_unlink (const unsigned short* f)	F	Parameter f is a pointer to a string value of a file name of an existing file	The file is deleted	Passed
152.	int f_unlink (const unsigned short* f)	F	Parameter f is a pointer to a string value of a file name of a non existing file	Nothing happens	Passed
153.	int f_unlink (const unsigned short* f)	F	Parameter f is a pointer to a string value of a directory name with no files	The directory is deleted	Passed

Test no.	Interface	Type	Description	Expected Result	Test Result
154.	int f_unlink (const unsigned short* f)	E	Parameter f is a pointer to a string value of a directory name with existing files	The directory is not deleted	Passed
155.	int f_mkdir (const unsigned short* f)	F	Parameter f is a pointer to a string value “newdir”	A new directory will be created	Passed
156.	int f_rename (const unsigned short* f, const unsigned short* n)	F	Parameter f is a pointer to a string value of a file name of an existing file. Parameter n is a pointer to a string value of a new file name	The original file is renamed to the new file name	Passed
157.	int f_open (FIL* fp, const unsigned short* fn, unsigned char b)	F	Parameter fp is a pointer to a FIL variable. Parameter fn is a pointer to string value of a file name of an existing file. Parameter b is a byte value of 1 (read mode)	Return 0 (success code)	Passed

Test no.	Interface	Type	Description	Expected Result	Test Result
158.	int f_open (FIL* fp, const unsigned short* fn, unsigned char b)	F	Parameter fp is a pointer to a FIL variable. Parameter fn is a pointer to string value of a file name of a non existing file. Parameter b is a byte value of 1 (read mode)	Return 4 (code for file not found)	Passed
159.	int f_open (FIL* fp, const unsigned short* fn, unsigned char b)	F	Parameter fp is a pointer to a FIL variable. Parameter fn is a pointer to a string value of a file name of a non-existing file. Parameter b is a byte value of 2 (write mode)	Return 4 (code for file not found)	Passed
160.	int f_open (FIL* fp, const unsigned short* fn, unsigned char b)	F	Parameter fp is a pointer to a FIL variable. Parameter fn is a pointer to a string value of a file name of a non-existing file. Parameter b is a byte value of 6 (create new and write mode)	Return 0 (success code)	Passed

Test no.	Interface	Type	Description	Expected Result	Test Result
161.	int f_close (FIL* fp)	F	Parameter fp is a pointer to a FIL variable previously returned by f_open	Return 0 (success code)	Passed
162.	int f_read (FIL* fp, void* b, unsigned int n, unsigned int* r)	F	Parameter fp is a pointer to a FIL variable previously returned by f_open. Parameter b is a pointer to a memory of size 20. Parameter n is a digit of value 20. Parameter r is a pointer to an unsigned integer variable.	Return 0 (success code)	Passed

Test no.	Interface	Type	Description	Expected Result	Test Result
163.	int f_write (FIL* fp, const void* b, unsigned int n, unsigned int* r)	F	Parameter fp is a pointer to a FIL variable previously returned by f_open. Parameter b is a pointer to a memory of size 20. Parameter n is a digit of value 20. Parameter r is a pointer to an unsigned integer variable.	Return 0 (success code)	Passed
164.	int f_lseek (FIL* fp, unsigned long s)	F	Parameter fp is a pointer to a FIL variable previously returned by f_open. Parameter s is a digit of value 1	Return 0 (success code)	Passed

F. ED APPLICATION TESTING

The objective of the following regression test is to verify that the modifications to the *ed* application have not impacted the functionality of the application in undesirable ways. This test is conducted in a Linux environment using the test suite distributed with the GNU *ed* source code. To run the test suite directly in the T-TASI system would require features that are currently not available in the T-TASI system, e.g., a shell and the *sed* utility. Table 14 provides a summary of the different tests that were conducted for the *ed* application.

Table 14. *ed* Application Test Case

Test no.	Interface	Type	Description	Expected Result	Test Result
165.	*ed* Application	F	To verify the modified ed application passes its regression test.	No error messages	Passed

G. INTEGRATION TESTING

The objectives of the following set of system-level tests are to verify that the previous software components (T-TASI C Library, T-TASI Application-Level Memory Management, *ed* Application and T-TASI RAM Disk File System) function correctly when run together on the T-TASI system. The system test involves four partitions (see Figure 7). Partition 1 (TPA partition) is configured with the trusted path application (TPA). Partition 1, Partition 2 (normal partition), Partition 3 (normal partition) and Partition 4 (EAP) each have a memory segment, specified in the configuration vector. These memory segments host a RAM disk file system for each of the partitions. The memory segment belonging to the partition is initialized by the file system to be the "0" drive (in Figure 7, each memory segment has the same color as the partition recognizing its RAM disk as the "0" drive). Inter-partition access flow is demonstrated by allowing

Partition 2 to have read and write access to the memory segment owned by Partition 3. Partition 3 is also allowed read and write access to the memory segment owned by Partition 2. Partitions recognize the RAM disks resident on those memory segments it does not own using other drive letters. For example, Partition 4 is allowed read and write access to the memory segment owned by Partition 2 (recognized as the "1" drive) and Partition 3 (recognized as the "2" drive). The memory segment owned by Partition 4 is accessible only to Partition 4, as it represents sensitive information that is only accessible to the EAP.

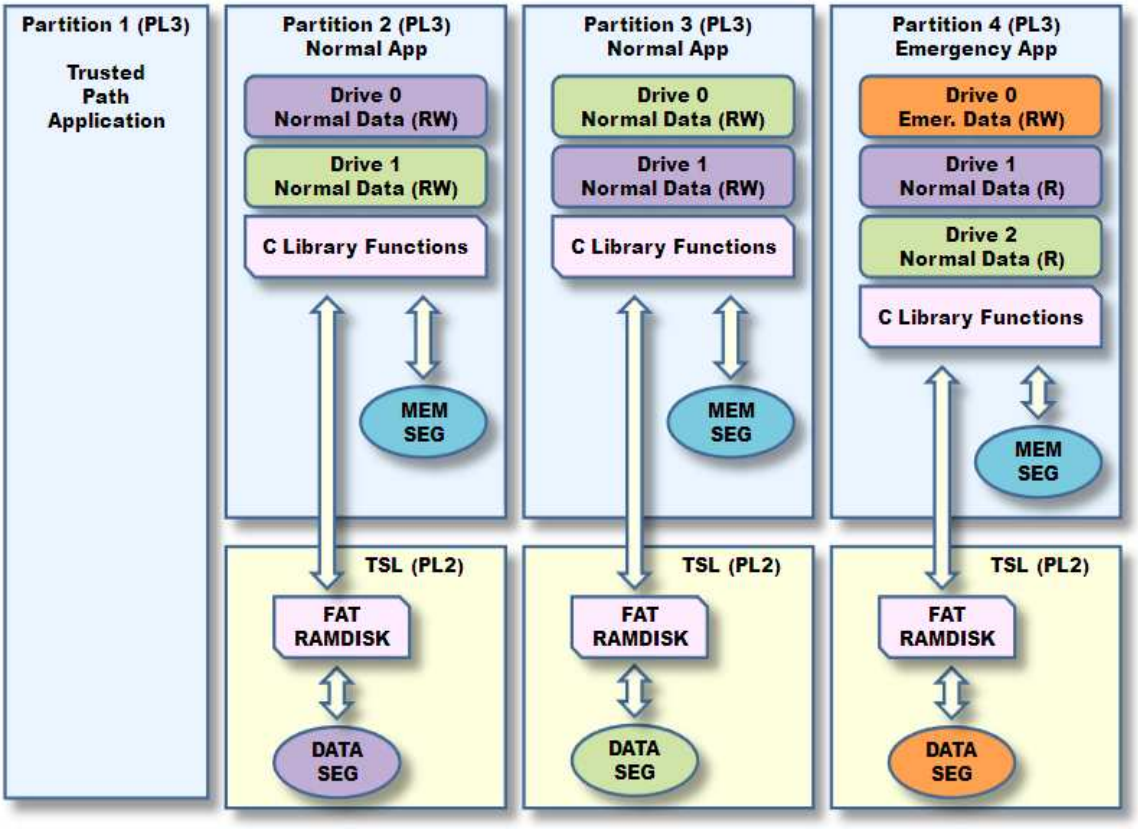

Figure 7. Setup for Integration Test

Table 15 provides a summary of the different tests conducted as part of integration testing.

Table 15. Integration Test Cases

Test no.	Access Mode	Type	Description	Expected Result	Test Result
166.	Normal Access	F	User creates a new file using ed application in partition 2 drive 0	A new file is created (Allowed internal partition flow)	Passed
167.	Normal Access	F	User creates a new file using ed application in partition 2 drive 1	A new file is created (Allowed external partition flow)	Passed
168.	Emergency Access	F	User creates a new file using ed application in partition 4 drive 0	A new file is created (Allowed internal partition flow)	Passed
169.	Emergency Access	F	User read a file using ed application in partition 4 drive 1	User is able to read the file (Allowed external partition flow)	Passed
170.	Emergency Access	E	User modifies a file using ed application in partition 4 drive 2	User is not able to modify the file (Disallowed external partition flow)	Passed

H. SUMMARY

This chapter described the test plans for the software components developed or ported in this project, described in detail in Chapter IV. All tests conducted were successful. The procedures for running the tests described in this chapter are provided in Appendix C. The following chapter discusses the general results of this project and suggests future work.

VI. RESULTS

In this thesis, we described our successful effort to port the *ed* text editor application to the T-TASI system, supporting a demonstration in which access to sensitive information is granted, under policy restrictions, during an emergency scenario. The software libraries and framework implemented for T-TASI application development will be useful to support future application development for the T-TASI project. In the following sections, we discuss some problems encountered during the course of our work, discuss related work, and conclude with suggestions for future work.

A. PROBLEMS ENCOUNTERED

1. Large Memory Array Initialization

When the LPSK was configured to provide a partition's application with a large static array of more than one megabyte, the T-TASI system halted upon startup with a memory error. The problem was traced to a bug in the LPSK. The kernel design utilized a per-partition *Local Descriptor Table* (LDT), which holds the PL1, PL2, and PL3 segment descriptors for the partition. During execution, only one LDT can be active (accessible) at a time. As the kernel initialized each partition, the kernel was not properly switching the LDT values for the new partition. Based on this discovery, the kernel bug was resolved very quickly.

2. Interface Name Conflicts

The T-TASI C library implements functions defined in the C99 standard, whose names conflict with existing utility functions provided by the T-TASI system's existing Application I/O Library API. The differences between the interfaces provided by the Application I/O Library and the identically named interfaces defined in the C99 standard are described in Table 16. In absence of a resolution to these symbol conflicts, the linker would fail during the application build process.

Table 16. Symbol Conflicts Between the T-TASI System Application I/O Library Interfaces and Standard C Library Interfaces

no.	Application I/O Library Interface	C Library Interface (C99)
1.	int __near strcpy(char* dest, const char* src, int len);	char* strcpy(char* dest, const char* src);
2.	int __near strncat(char* dest, const char* src, int len);	char* strncat(char* dest, const char* src, unsigned int len);
3.	void __near memcpy(unsigned char* dest, unsigned char const* src, int len);	void* memcpy(void* dest, const void* src, unsigned int len);
4.	int __near memcmp(unsigned char const* addr_1, unsigned char const* addr_2, int len);	int memcmp(const void* addr_1, const void* addr_2, unsigned int len);

The work-around for development was to comment out the conflicting interface names in the Application I/O Library header file. This allows the linker to proceed and does not affect the compilation of the rest of the system. We suggest developers avoid

choosing interface names that exist in standard library packages and suggest exported interfaces in the T-TASI system be renamed with prefixes corresponding to module name or PL number.

3. Identifier for Memory Segment Declaration

Memory segments were used for different purposes, such as memory management and file systems, in this work. In the LPSK configuration vector, the path field of a declared data segment can be used by a partition's application to directly reference the data segment. However, there is no identifier field associated with a memory segment in the LPSK configuration vector. Thus, in order to differentiate the usage of different memory segments, we developed a convention in which the size of the memory segment can be used to infer that segment's function. In particular, the size of a memory segment used for memory management is hard-coded in the T-TASI Application-Level Memory Management, and the size of a memory segment holding a RAM disk is hard-coded in the T-TASI RAM Disk File System. This is not ideal and future work on the T-TASI system could incorporate an identifier field representing the type for a memory segment.

4. User Credentials in Non TPA Partition

When the T-TASI system is booted, a user has to authenticate to the system using a user name and password. This login mechanism is present only in the TPA partition and not available in other partitions. There is no authentication mechanism in the new partition when a user changes focus to a non-TPA partition. This would not be a problem if the current user’s name (as input to the TPA partition) can be retrieved from the Trusted OS Service, e.g., using the *scos_who* interface, by other partitions. An application in a non-TPA partition sometimes requires the current user's name for purposes such as application-level authorization. In particular, if the E-device is shared with other users, another authorization mechanism may be required for the EAP to restrict certain unauthorized users from accessing sensitive information.

B. RELATED WORK

The STOP operating system and XTS systems (e.g., XTS-200, XTS-300) are descendents of the SCOMP security kernel. In particular, the STOP 6.3 operating system is a commercially available, multilevel secure, high-assurance operating system that was evaluated, as part of the XTS-400 system [32], to meet the Common Criteria assurance level rating of EAL5+. The STOP 7 operating system, unlike previous STOP operating systems, provides UNIX-like features (a shell, utilities, etc) and a standard C library (a port of the open-source library uClibc) to develop or port UNIX applications to run on their security kernel. This is similar to the larger objectives of the software components developed during this project, which provide simple interfaces for porting and developing applications on the T-TASI system. Recall, however, that the LPSK is intended to be an evaluated, high-assurance separation kernel, satisfying the Separation Kernel Protection Profile.

C. FUTURE WORK

In this section, we suggest future work related to improving the application development framework and software components developed in this project, for the T-TASI system.

1. T-TASI C Library

Some of the standard C interfaces have not been implemented by our effort because of a lack of supporting functionalities provided by the current T-TASI system. Table 17 shows the interfaces in a standard C library and the corresponding required system functionalities.

Table 17. C Interfaces and the Corresponding Required System Functionalities

no.	C Interface	Required System Functionalities
1.	Standard Signal Handling interfaces (e.g., *kill*, *sigaction*)	Provide standard signal such as SIGINT (Interactive attention signal), SIGILL (Illegal instruction), SIGABRT (Abnormal termination), SIGTERM (Termination request). A signal can report some exceptional behavior (divide by zero) within the program, or a signal can be used by the system to report some asynchronous event (user pressing a break key). A signal handling API is currently provided in the T-TASI system, but it does not define or utilize standard POSIX signals.
2.	Process spawning interfaces (e.g., *fork, vfork, exec*)	Provide a process a means to spawn a child process. This feature requires the LPSK to support more than one process per partition. It would be necessary to create a process manager in the Trusted or Untrusted Operating System Services.
3.	Shell interaction interfaces (e.g., *system*, *popen*, *pathconf*)	Provide a process access to a shell (e.g., bash, csh). These interfaces rely on a shell interpreter existing. Again, before a shell interpreter can be ported to the T-TASI system, the LPSK will need to support multiple processes per partition , and operating system services would be required to give the illusion of process creation and termination.

2. T-TASI RAM Disk File System

The file storage implemented in this thesis is not persistent across a power cycle, as the RAM disk is, of course, implemented in memory. The FatFs software library, used in this work, does implement support for a variety of persistent storage. There are existing open source projects using FatFs to utilize physical storage media such as Secure Digital (SD) memory card storage, Multimedia Card (MMC) storage and USB flash storage [25]. Low-level device communication codes for these media are available in those projects, thus reducing future development time. Extending FatFs support to include IDE or SATA disk support may benefit an effort to integrate non-bootable raw secondary storage devices with the T-TASI system.

3. T-TASI Application-Level Memory Management

Lister and Eager [33] state the following five requirements for memory management in an operating system:

1. Relocation: Memory management should be able to relocate programs in memory, and handle both memory references and addresses in relocated program code so that they always point to the correct memory location.
2. Protection: Each process should be protected against accidental or malicious interference by other processes.
3. Sharing: Any protection mechanism should have the flexibility to allow several processes to access the same region of memory and share information.
4. Logical Organization: Memory management should be able to logically differentiate different parts of memory such as execute only, read only or read / write only.
5. Physical Organization: Memory Management should handle moving information between main and secondary memory.

The memory management provided in the T-TASI system satisfies all but the last requirement (i.e., swapping memory into secondary memory). Although the allocation of

memory segments to each partition is static, a low-level secondary storage device driver along with a low-level swap manager could provide the capability to swap these static memory segments between secondary storage and RAM dynamically. Swapping is part of the LPSK design [10]; however, it has not yet been implemented.

D. CONCLUSION

The application development framework presented here, consisting of the T-TASI C Library, the T-TASI Application Memory Management and the T-TASI RAM Disk File System, was designed and implemented to meet the requirements of supporting extraordinary access of sensitive information during an emergency on the T-TASI system. The open source text editor *ed* was ported using this framework, to demonstrate the ability of the T-TASI system to utilize a non-trivial application in an emergency partition.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. DESIGN OVERVIEW OF FATFS

This appendix provides a description of the FatFs module and the interfaces exported by the T-TASI RAM Disk File System.

A. FATFS RETURN CODES

Table 18 provides the list of possible return codes returned by the T-TASI RAM Disk File System functions that are exported from PL2 to PL3. These return codes are from the FatFs project. The second column refers to the type of return code. The third column refers to the actual value of the return code. The fourth column gives a description of the meaning of the return code.

Table 18. FatFs Function Return Codes

no.	Type	Value	Meaning
1.	FR_OK	0	Success
2.	FR_DISK_ERR	1	A hard error had occurred in the low level disk I/O layer
3.	FR_INT_ERR	2	Assertion failed
4.	FR_NOT_READY	3	The physical drive cannot work
5.	FR_NO_FILE	4	Could not find the file
6.	FR_NO_PATH	5	Could not find the path
7.	FR_INVALID_NAME	6	The path name format is invalid

no.	Type	Value	Meaning
8.	FR_DENIED	7	Access denied due to prohibited access or directory full
9.	FR_EXIST	8	Access denied due to prohibited access
10.	FR_INVALID_OBJECT	9	The file/directory object is invalid
11.	FR_WRITE_PROTECTED	10	The physical drive is write protected
12.	FR_INVALID_DRIVE	11	The logical drive number is invalid
13.	FR_NOT_ENABLED	12	The volume has no work area
14.	FR_NO_FILESYSTEM	13	There is no valid FAT volume on the physical drive
15.	FR_MKFS_ABORTED	14	The f_mkfs() is aborted due to parameter error
16.	FR_TIMEOUT	15	Could not get a grant to access the volume within defined period
17.	FR_LOCKED	16	The operation is rejected according to the file sharing policy

no.	Type	Value	Meaning
18.	FR_NOT_ENOUGH_CORE	17	Working buffer for the long file name could not be allocated
19.	FR_TOO_MANY_OPEN_FILES	18	Too many files opened

B. FATFS FILE MODES

Table 19 provides the list of possible modes for the *f_open* interface, exported by the T-TASI RAM Disk File System to PL3. These numerical values of the file modes are from the FatFs project.

Table 19. FatFs File Open Modes

no.	Modes	Meaning
1.	0	Opens the file. The function fails if the file does not exist
2.	1	Specifies read access to the object. Data can be read from the file
3.	2	Specifies write access to the object. Data can be written to the file
4.	4	Creates a new file. The function fails if the file already exists
5.	8	Creates a new file. If the file already exists, it is truncated and overwritten
6.	16	Opens the file if the file exists. If not, the function creates the new file

C. FATFS FIL STRUCTURE

Table 20 provides details for the member variables of the internal file structure FIL (see Table 6), defined by the FatFs project.

Table 20. FatFs FIL Structure

no.	Item	Used for
1.	fs	Pointer to the owner file system object.
2.	id	The id of the file system mounted.
3.	flag	Flags used when the file is opened.
4.	pad1	Padding character
5.	fptr	Read and write file pointer.
6.	fsize	The size of the opened file
7.	ori_clust	The cluster where the file starts.
8.	curr_clust	The current cluster of the file where the fptr is referencing.
9.	dsect	The current data sector of the file.
10.	dir_sect	The sector containing the directory entry.
11.	dir_ptr	The pointer to the directory entry in the window.

D. FATFS INTERFACES

This section describes the PL2 functions exported by the T-TASI RAM Disk File System to PL3. The interfaces are provided by the FatFs project. They were modified with the appropriate call gate specification (CALLGATE_DECL_SCOS) needed to export these interfaces to PL3. These interfaces are *f_open, f_read, f_write, f_close, f_unlink, f_mkdir, f_rename, f_lseek* (see Table 4).

The following are details for each of these functions.

1. f_open

This function creates a file object (FIL) to be used to access a file.

1.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_open(

FIL* *fptr*,
const TCHAR* *fn*,
BYTE *mode*)

1.2 Inputs

- *fptr*
  [OUT] Return the pointer to the file object structure to be created. After the *f_open* function succeeded, the file can be accessed with the file object structure until it is closed

- *fn*
  [IN] Pointer to a null-terminated string that specify the file name to create or open

- *mode*
  [IN] Specify type of access and open method for the file. It is specified by a combination of the flags listed in Table 19

1.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18. On success, the *fptr* will be pointing to a valid FIL structure.

2. f_read

This function reads data from an opened file.

2.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_open(
FIL* *ftr*,
void* *buffer*,
UINT *bytestoread*,
UINT* *bytesread*)

2.2 Inputs

- *fptr*
  [IN] The pointer to the opened file object

- *buffer*
  [OUT] Pointer to a buffer to store the read data

- *bytestoread*
  [IN] Specify the number of bytes to read from the file

- *bytesread*

[OUT] Pointer to an unsigned integer to return the actual number of bytes read from the file

2.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

3. f_write

This function writes data to an opened file.

3.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_write(
FIL* *fptr*,
const void* *buffer*,
UINT *bytestowrite*,
UINT* *byteswritten*)

3.2 Inputs

- *fptr*
  [IN] The pointer to the opened file object

- *buffer*
  [IN] Pointer to a buffer that stores the data to be written

- *bytestowrite*
  [IN] Specify the number of bytes to write to the file

- *byteswritten*
  [OUT] Pointer to an unsigned integer to return the actual number of bytes written to the file

3.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

4. f_close

This function closes an opened file.

4.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_close(
FIL* *fptr*)

4.2 Inputs

- *fptr*
  [IN] The pointer to the opened file object

4.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

5. f_unlink

This function removes a file or directory from the file system.

5.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_unlink(
const TCHAR* *buffer*)

5.2 Inputs

- *buffer*
  [IN] Pointer to a null terminated string that specifies the file (or directory) to be removed. A non-empty directory cannot be removed with this function

5.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

6. f_mkdir

This function creates a directory in the file system.

6.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_mkdir(
const TCHAR* *buffer*)

6.2 Inputs

- *buffer*
  [IN] Pointer to a null terminated string that specifies the name of the directory to be created

6.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

7. f_rename

This function renames a file in the file system.

7.1 Prototype

__CALLGATE_DECL_SCOS FRESULT f_rename(
const TCHAR* *oldname*,
const TCHAR* *newname*)

7.2 Inputs

- *oldname*
  [IN] Pointer to a null terminated string that specifies the name of the file to be renamed

- *newname*
  [IN] Pointer to a null terminated string that specifies the new name of the file to be renamed

7.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

8. f_lseek

This function moves the file read/write pointer of an opened file.

8.1 Prototype

```
__CALLGATE_DECL_SCOS FRESULT f_lseek(
        const FIL* fptr,
        DWORD offset)
```

8.2 Inputs

- *fptr*
  [IN] The pointer to the opened file object

- *offset*
  [IN] Number of bytes from the start of the file

8.3 Function Result

Return FR_OK on success or any of the failure codes in Table 18.

APPENDIX B. INSTALLATION GUIDE

This appendix describes the installation procedure for the T-TASI system and those for the software artifacts of this project. Section A describes the system requirements prior to installation and Section B describes the procedures for setting up the virtual machines.

A. SYSTEM REQUIREMENTS

The following hardware and software are required before proceeding to the installation. These were used during the implementation of this thesis.

The VM host machine: a desktop machine with the following, or comparable, hardware and software configuration:

Intel Core2 Quad CPU, 3 GHz. 4 GB RAM

Windows XP Professional (Service Pack 3) Operating System

VMWare Workstation 7.1.0

CISR Archive ID *Thesis-ed-2010*, Disc 1 (of 4)

Contains the primary software components developed as part of this project: T-TASI C Library, T-TASI RAM Disk File System, T-TASI Application-Level Memory Management, modified *ed* source code.

Also, contains updated T-TASI project kernel code.

CISR Archive ID *Thesis-ed-2010*, Disc 2 (of 4) and Disc 3 (of 4)

Contains the T-TASI development platform VM.

CISR Archive ID *Thesis-ed-2010*, Disc 4 (of 4)

Contains the T-TASI System VM.

B. SYSTEM INSTALLATION

The following steps will set up the development platform virtual machine (VM) and the T-TASI system VM.

Copy all files on Disc 2 and Disc 3 to a new and empty folder on the host machine.

Import the T-TASI development platform VM into VMware by opening the file "Red Hat Linux.vmx" in the folder from step 1.

Copy all files on Disc 4 to another new and empty folder on the host machine.

Import the T-TASI System VM into VMware by opening the file "Red Hat Linux.vmx" in the folder from step 3.

Boot the T-TASI development platform VM using the option, "Power on this virtual Machine", in VMWare.

Login to the development platform VM using user name "student" and password "Password1".

On the development platform:

Copy the tarballs *thesis_dev.tgz* and *thesis_test.tgz* from Disc 1 to the local path ~ */Documents/tcx/trunk/*

Extract all files from the tarball:

cd ~/Documents/tcx/trunk/

tar –xzvf thesis_dev.tgz

Compile a new version of the T-TASI System kernel and applications:

cd ~/Documents/owc_17

. ./setvars.sh

cd ~/Documents/tcx/trunk/kernel

make clean all

Compile a new version of the configuration vector:

cd ~/Documents/tcx/trunk/vector/src

make clean all

./create_vector

Create a tarball of the T-TASI System binaries:

cd ~/Documents/tcx/trunk

./tarit

Boot the T-TASI System VM using the option, “Power on this virtual Machine”, in VMWare.

When the Grub boot loader menu is presented, select the first option: “Fedora (2.6.23.17-88.fc7)”.

Login to Fedora using the user name “root” (password is not required).

On the Fedora machine running on the T-TASI System VM:

Copy the file *thesis_test.tgz* from the development VM to the local root directory (substitute the IP address of the development VM for the *IP*. The IP address can be obtained using the command, */sbin/ifconfig*):

scp
student@*IP*:~/Documents/tcx/trunk/thesis_test.tgz
/

Extract all files from the tarball:

cd /

tar –xzvf thesis_test.tgz

Edit the script */root/install.sh* to reflect the IP address of the development platform virtual machine.

Update the files on the T-TASI System using this script:

cd /root

./install.sh

When prompted by the install script, the password is “Password1”.

When the installation is completed, Fedora will reboot.

Upon reboot, at the Grub boot menu, select “LPSK with all Applications”.

The system will boot to the T-TASI system.

APPENDIX C. TESTING PROCEDURES

This appendix provides details on the procedures to perform the tests described in Chapter V. The installation step in Appendix B should be followed before starting the test procedures described next.

A. TEST PROCEDURES FOR TEST CASE 1–163

The following steps will perform the unit tests for test cases 1–163:

Boot the T-TASI System VM.

Log in to the T-TASI System using the user name “user1” and password “Password1”.

Change partition focus to the (normal) Partition 2.

Start the test session, using the command:

test

The following message will appear on the screen:

Test Start

The test case 134 is successful if the following message appears on the screen:

Hello World

The test case 135 is successful if the following message appears on the screen:

1234

The test case 136 is successful if the following message appears on the screen:

z

For test case 141, the tester will be prompted to type a string, and the user should do so when prompted.

For test case 142, the tester will be prompted to type a string, and the user should do so when prompted. The test case will repeat until the correct string is typed.

For test case 143, the tester will be prompted to type a string, and the user should do so when prompted. The test case will repeat until the correct string is typed.

The test is complete when the following message appears on the screen:

Test End

All tests have completed successfully if the following message appears on the screen:

All tests completed successfully

If any test completes with a failure, the following message appears on the screen:

Some tests failed

Type the following command to clear the temporary files created during testing (this step is necessary if repeated testing will be performed):

cleartest

B. TEST PROCEDURES FOR TEST CASE 164

The following procedures can be used to perform the test for test case 164. The following steps test the original *ed* application in a Linux system using its pre-packaged test suite:

Boot the development platform VM.

Log in to the development platform using user name “student” and password “Password1”.

Compile and prepare the original *ed* application for testing:

cd ~/Documents/tcx/trunk/kernel/ed

tar –xzvf ed-1.4.tgz

cd ed-1.4

./configure

make

cd testsuite

cp ../ed .

Type the following commands to run the *ed* application's test suite:

./check.sh

The following message appears on the screen if all tests complete successfully:

tests completed successfully

The following steps test the modified *ed* application in a Linux system using its pre-packaged test suite:

Compile and prepare the modified *ed* application for testing:

cd ~/Documents/tcx/trunk/kernel/ed

./compile

cd ed-1.4/testsuite

rm ed

cp ../../ed .

Type the following commands to test the modified *ed* application:

./check.sh

The following message appears on the screen if all tests complete successfully:

tests completed successfully

C. TEST PROCEDURES FOR TEST CASE 165–169

The following procedure will perform the Integration test cases 165–169. First, the following steps perform the test for test case 165.

Boot the T-TASI System VM.

Log in to the T-TASI System using user name “user1” and password “Password1”.

Change partition focus to (normal) Partition 2.

Create a new file on disk 0 of Partition 2:

ed newfile1

i

Data in the text file

.

w

q

Verify the file *newfile1* was created correctly:

ed newfile1

1p

q

The test is successful so far if the following message appears:

Data in the text file

Use the SAK to switch to the TPA partition, then switch focus to (normal) Partition 3.

Verify that the file *newfile1* was created on disk 1 of Partition 3:

ed 1:newfile1

1p

q

The test completed successfully if the following message appears:

Data in the text file

The following steps will perform tests for test case 166:

Use the SAK to switch to the TPA partition, then switch focus to (normal) Partition 2.

Create a new file on disk 1 of Partition 1:

ed 1:newfile2

i

Data in the 2nd text file

.

w

q

Verify the file *newfile2* was created correctly:

ed 1:newfile2

1p

q

The test is successful so far if the following message appears:

Data in the 2nd text file

Use the SAK to switch to the TPA partition, then switch focus to (normal) Partition 3.

Verify that the file *newfile2* was created on disk 0 of Partition 3:

ed 0:newfile2

1p

q

The test completed successfully if the following message appears:

Data in the 2nd text file

The following steps will perform tests for test case 167:

Use the SAK to switch to the TPA partition, then switch focus to (EAP) Partition 4.

Create a new file on disk 0 of Partition 4:

ed secret1

i

Secret data in file

.

w

q

Verify the file *secret1* was created correctly:

ed 0:secret1

1p

q

The test completed successfully if the following message appears:

Secret data in file

The following steps will perform tests for test case 168:

Open and read the file *newfile1* on disk 1 of Partition 4 (i.e., a file owned by Partition 2):

ed 1:newfile1

1p

q

The test completed successfully if the following message appears:

Data in the text file

The following steps will perform tests for test case 169:

Open and modify the file *newfile2* on disk 2 of Partition 4 (i.e., a file owned by Partition 3):

ed 2:newfile2

1,2p

1p

a

New data added to this file

.

1,2p

w

Q

The test is successful so far if the following message appears:

Data in the 2nd text file

New data added to this file

Check that the file *newfile2* was not actually modified:

ed 2:newfile2

1,2p

1p

q

The test is successful if the following message appears:

Data in the 2nd text file

APPENDIX D. DEMONSTRATION PROCEDURES

This appendix documents the procedures to demonstrate the capability of the T-TASI system and the underlying LPSK to provide transient trust from a normal operating mode to an emergency mode. Section A describes the preparation required in advance of running the demonstration. Sections B–D describe the main scenarios of the demonstration.

A. PREPARATION

The following steps will prepare the T-TASI system for the demonstration. All commands issued are to be followed by a new line character.

1. Boot up the T-TASI System VM.
2. Log in to the T-TASI System using the username “user1” and password “Password1”.
3. Toggle the emergency mode “on” in the TPA partition using the command:

   ```
   T
   ```

4. Change partition focus to the (EAP) Partition 4, using the command:

   ```
   3
   ```

5. Prepare the demonstration session for the EAP using the command:

   ```
   demo
   ```

6. Use the SAK to switch to the TPA partition. Then change partition focus to the (normal) Partition 2, using the command:

   ```
   1
   ```

7. Prepare the demonstration session for the normal partitions using the command:

   ```
   demo
   ```

8. Use the SAK to switch to the TPA partition. Then, toggle the emergency mode back to “off” using the command:

   ```
   T
   ```

9. Return to the TPA main screen using the command:

   ```
   c
   ```

The system is now ready for demonstration.

B. SCENARIO: ACCESSING NORMAL PARTITION IN NORMAL MODE

The following will demonstrate the capability of the T-TASI system to support a normal mode of operation. The preparation steps in Section A needs to precede before this section.

1. The T-TASI system is currently in the normal mode operation. Switch to the focus menu, using the command:

   ```
   F
   ```

   Notice, in normal mode operation there are three visible partitions: Partition 1 is used for the trusted path application, while Partitions 2 and 3 are normal partitions. Partition 4 cannot be seen, because it is the EAP and we are in a normal mode.

2. Switch to (normal) partition 2, using the following command:

   ```
   1
   ```

3. In Partition 2, there is a file containing some (fictitious) non-sensitive patient records. Display these records with *ed* using the commands:

   ```
   ed rec_nor.txt
   1,$ p
   ```

4. Users can modify or update the records in this file. Modify the phone number of a patient using the following commands:

   ```
   s/831-3737271/831-3737279/g
   ```

```
1,$ p
wq
```

C. SCENARIO: ACCESSING EAP IN EMERGENCY MODE

The following will simulate the capability of the T-TASI system to operate in an emergency mode, demonstrating an application running in the EAP.

1. The T-TASI system receives a signal, triggering the activation of the emergency. Currently, this functionality is not available and thus we simulate this behavior manually. Use the SAK to switch to the TPA and toggle emergency mode “on” using the command:

   ```
   T
   ```

2. Notice, in emergency mode, the EAP is now visible. Switch focus to the EAP, using the command:

   ```
   3
   ```

3. The EAP contains sensitive patient records not available during normal mode operation. For this demo, our (simulated) “sensitive data” includes details about allergies and social security numbers for each patient. Type the following commands to display the “sensitive data” for each patient:

   ```
   ed rec_sec.txt
   1,$ p
   q
   ```

D. SCENARIO: ACCESSING NORMAL PARTITION IN EMERGENCY MODE

The following will simulate the capability of the T-TASI system to operating in an emergency mode, demonstrating an application running in the EAP. In particular, we demonstrate that EAP applications are allowed read-only to access normal partition data.

1. In the EAP, display the normal patient records using the command:

```
ed 1:rec_nor.txt
1,$ p
```

2. Now, try to modify information in the normal partition by editing the phone number:

```
s/831-3737279/831-3737271/g
1,$ p
wQ
```

3. Display the file again, using the commands:

```
ed 1:rec_nor.txt
1,$ p
q
```

Notice, the previous edits were not saved.

4. When the emergency situation is over, the T-TASI system will receive another signal to deactivate the emergency mode. Again, this functionality is not yet available, so it is simulated using the following procedure. Use the SAK to switch to the TPA, and toggle emergency mode “off” using the command:

```
T
```

Notice, again, the EAP is not visible to users and cannot receive focus.

APPENDIX E. SOFTWARE ARTIFACTS

This appendix provides information on the software artifacts modified or created during the course of this research work. Section A provides a list of files and source code related to building the *ed* text editor. Section B provides a list of files used in the implementation of the T-TASI C Library. Section C provides a list of files used in the implementation of the T-TASI RAM Disk File System. Section D provides a list of the original T-TASI system files that were modified during the course of this work.

A. ED APPLICATION

This section lists the files used for building the *ed* text editor (see Table 21). The files listed in this section can be found in the sub-directory /*ed* in the main kernel directory. Column 2 shows the file name and column 3 provides a brief description of the purpose of the file.

Table 21. Files Used to Build the *ed* Text Editor on the T-TASI System

o.	File Name	Purpose
	ed_app.c	Wrapper application to start *ed* application in the T-TASI system
	scos_misc_nyc.h	A copy of the original *scos_misc.h* file with conflicting interface names removed.
	ed/ed.h	GNU *ed* header file
	ed/ed_main.c	Modified GNU *ed main.c*
	ed/ed-1.4.tgz	Original GNU *ed* application source code. This is used for testing the modified *ed* source code
	ed/compi	Script for compiling the modified *ed* application in a

	le	Linux development platform

B. T-TASI C LIBRARY

This section lists the files used to build the T-TASI C Library (see Table 22). The files listed in this section are found in the sub-directory /*ed* in the main kernel directory. Column 2 shows the file name and column 3 provides a brief description of the purpose of the file.

Table 22. Files Used to Build the T-TASI C Library

o.	File Name	Purpose
	clib.c	Source code for implementing the C library. It also contains the implementation of the T-TASI Application-Level Memory Management component
	stdio.h	Header file used for the T-TASI C Library
	regex/regcomp.c	FreeBSD regular expression parsing code
	regex/regfree.c	FreeBSD regular expression parsing code
	regex/regexec.c	FreeBSD regular expression parsing code
	regex/regerror.c	FreeBSD regular expression parsing code

C. T-TASI RAM DISK FILE SYSTEM

This section lists the files used to build the T-TASI RAM Disk File System library (see Table 23). The files listed in this section are found in the sub directory /*pl2fs*

in the main kernel directory. Column 2 shows the file name and column 3 provides a brief description of the purpose of the file.

Table 23. Files Used to Build the T-TASI RAM Disk File System

o.	File Name	Purpose
	diskio.h	Header file used for *disk_io.c*
	diskio.c	Implements the low level disk I/O layer for the file system. The interface with a memory segment to create a RAM disk is found in this source code
	ff.h	Header file used for *ff.c*
	ff.c	Implements the FAT12/16/32 file system
	ffconf.h	Configuration file for the FatFs file system. Settings such as sector size can be set in this file. The default settings from FatFs project are used for this work

D. MODIFICATION OF ORIGINAL T-TASI SYSTEM SOURCE CODE

This section lists the files that were modified during this development work (see Table 24) . All files listed below can be found in the main kernel directory. Column 2 shows the file name and column 3 provides a brief description of the modification to the original file.

Table 24. T-TASI System Files that were Modified

o.	File Name	Modification Purpose

o.	File Name	Modification Purpose
	tpa_app.c	Included codes for simulating receipt of a start/end emergency signal and hiding/displaying the EAP partition
	scos_gates.h	Included T-TASI RAM Disk File System in the PL2 call gates
	scos.c	Included a line to initialize the memory segment for the T-TASI RAM Disk File System
	Makefile	Included lines for compiling the *ed* application and compiling the T-TASI RAM Disk File System into PL2

APPENDIX F. BASIC ED COMMANDS

This appendix provides summary instructions for basic commands to operate *ed*. The details in this appendix are an abridged version of instructions found in the *ed* man pages. For each row in Table 25, the second column provides the *ed* command, the third column provides a brief description of the command and the final column provides a description and example of how to use the command.

Table 25. Basic *ed* Commands

no.	Command	Purpose	Usage
1.	`e`	Edit file	`e` *filename* Open and read the file specified by *filename*.
2.	`d`	Delete lines	*n* `d` Delete line number *n* in the file. *n1*,*n2* `d` Delete the range of lines *n1* to *n2* in the file.
3.	`i`	Insert lines	`i` Start " insert mode" in ed. When in this mode, text will be inserted after the addressed line. Typing "`.`" on a new line will end insert mode.

no.	Command	Purpose	Usage
4.	`a`	Append lines	`a` Start “append mode” in *ed*. When in this mode, text can be entered after the addressed line. Typing “.” on a new line will end append mode.
5.	`p`	Print lines	*n* `p` Print the line number *n* of the file. *n1*,*n2* `p` Print line number *n1* to *n2* of the file. The range *1*,*$* with print, will print the entire file.
6.	`t`	Copy lines	*n1*,*n2* `t` *n3* Copy lines from *n1* to *n2* to the line after *n3*.
7.	`m`	Move lines	*n1*,*n2* `m` *n3* Move lines from *n1* to *n2* to the line after *n3*.
8.	`w`	Write file	`w` *filename* Write data in the buffer to the file named *filename*. If there is no file name specified, the current file will be overwritten.

no.	Command	Purpose	Usage
9.	`u`	Undo last command	`u` Undo the last command that modified the file.
10.	`q`	Quit *ed*	`q` Causes *ed* to exit. If there are changes in the file, the user will be notified.
11.	`Q`	Quit *ed*	`Q` Causes *ed* to exit without checking whether there are changes in the file.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] J. P. Anderson, "Computer Security Technology Planning Study," *Air Force Elec. Syst. Div. Rep. ESD-TR-73-51*, October 1972.

[2] J. H. Saltzer and M.D. Schroeder, "The protection of information in computer systems," in *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, September 1975.

[3] W. R. Shockley and R. R. Schell, "TCB subsets for incremental evaluation," in *Proceedings of the 3rd AIAA Conference on Computer Security*, pp. 131–139, 1987.

[4] S.R. Jr. Ames, M. Gasser and R.R. Schell, "Security kernel design and implementation: an introduction," Computer, vol. 16, no. 7, pp. 14–22, July 1983.

[5] L. J. Fraim, "SCOMP: a solution to the multilevel security problem," in *Advances in Computer System Security*, vol. 2, Ed. R. Turn, pp. 185–93, 1983.

[6] R. R. Shell, T. F. Tao and M. Heckman, "Designing the GEMSOS security kernel for security and performance," in *Proceedings of the 8th National Computer Security Conference*, pp. 108–19, 1985.

[7] J. Rushby, "The design and verification of secure systems," *8th ACM Symposium on Operating System Principles (SOSP)*, Pacific Groove, CA, Appears in *ACM Operating Systems Review*, vol. 15, no. 5, pp. 12–21, December 1981.

[8] T.D. Nguyen, T.E Levin and C.E. Irvine, "High Robustness Requirements in a Common Criteria Protection Profile," in *Proceedings of the 4th IEEE International Information Assurance Workshop*, April 2006.

[9] T. E. Levin, C. E. Irvine and T. D. Nguyen, "Least privilege in separation kernels," in *Proceeding of the International Conference on Security and Cryptography*, pp. 355–362, August 2006.

[10] P.C. Clark, D. J. Shifflett, C. E. Irvine, T. D. Nguyen and T. E. Levin, "Trusted Computing Exemplar (TCX) Least Privilege Separation Kernel (LPSK) Product Functional Specification Volume I High Level Description," 6 May 2010.

[11] National Security Agency. "U.S. Government Protection Profile for Separation Kernels in Environments Requiring High Robustness," Version 1.03, 29 June 2007.

[12] T. D. Nguyen, T. E. Levin and C. E. Irvine, "TCX project: high assurance for secure embedded systems," in *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 21–25, March 2005.

[13] Common Criteria for Information Technology Security Evaluation, Part 3: Security Assurance Requirements, Version 2.1, August 1999.

[14] T.E Levin, C.E. Irvine, T.V. Benzel, T.D. Nguyen, P.C. Clark and G. Bhaskara, " Trusted Emergency Management," NPS Technical Report NPS-CS-09-001, Naval Postgraduate School, Monterey, CA, March 2009.

[15] C.E Irvine, T.E Levin, P.C Clark and T.D. Nguyen, "A security architecture for transient trust," in *Proceedings of 2nd ACM Workshop on Computer Security Architectures* (CSAW '08), FairFax, VA, pp. 1–8, October 2008.

[16] Department of Defense Trusted Computer System Evaluation Criteria. no. DoD 5200.28–STD, National Computer Security Center, December 1985.

[17] J. Guillen, "Least Privilege Separation Kernel Storage Hierarchy Prototype for the Trusted Computing Exemplar Project," M.S. Thesis, Naval Postgraduate School, Monterey, CA, June 2010.

[18] The GNU Project, "GNU Emacs," http://www.gnu.org/software/emacs/. Last Accessed: December 6, 2010.

[19] Vim, "Vim Online," http://www.vim.org/. Last Accessed: December 6, 2010.

[20] The GNU Project , "Ed – A line-oriented text editor," http://www.gnu.org/software/ed/. Last Accessed: December 6, 2010.

[21] T. E. Levin, C. E. Irvine, T. V. Benzel, G. Bhaskara, P. C. Clark and T. D. Nguyen, "Design Principles and Guidelines for Security," Technical Report NPS-CS-07-014, Naval Postgraduate School, Monterey, CA, November 2007.

[22] C. Bays, "A comparison of next-fit, first-fit and best-fit," *Communications of the ACM*, vol. 20, no. 3, pp. 191–192, 1977.

[23] A. Tanenbaum, *Operating Systems Design and Implementation* (Prentice Hall, 2006), p. 382.

[24] ISO/IEC 9899:TC3 Draft Standard – Programming Languages – C, http://www.open-std.org/JHTC1/SC22/WG14/www/docs/n1256.pdf. Last Accessed: December 7, 2010.

[25] FatFs Project, "FatFs Generic File System Module," http://elm-chan.org/fsw/ff/00index_e.html. Last Accessed: July 2010.

[26] FatFs Project, “About FatFs License,” http://elm-chan.org/fsw/ff/en/appnote.html#license. Last Accessed: July 2010.

[27] GNU, “GNU C Library,” http://www.gnu.org/software/libc/. Last Accessed: July 2010.

[28] F. V. Leitner, “diet libc - A libc Optimized for Small Size,” http://www.fefe.de/dietlibc/. Last Accessed: July 2010.

[29] E. Andersen, “uClibc,” http://www.uclibc.org/. Last Accessed: July 2010.

[30] FreeBSD, “The FreeBSD Project,” http://www.freebsd.org/. Last Accessed: July 2010.

[31] Open Source Initiative OSI, “The BSD License,” http://www.opensource.org/licenses/bsd-license.php. Last Accessed: July 2010.

[32] “NIAP Common Criteria Evaluation and Validation Scheme Validated Product: XTS-400/STOP 6.4 U4,” Validation Report CCEVS-VR-VID10293-2008, http://www.niap-ccevs.org/cc-scheme/st/vid10293. Last Accessed: December 7, 2010.

[33] A. Lister and R. Eager, *Fundamentals of Operating Systems* (New York: Springer-Verlag), 1993.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, VA

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, CA

3. Kris Britton
   National Security Agency
   Fort Meade, MD

4. John Campbell
   National Security Agency
   Fort Meade, MD

5. Deborah Cooper
   DC Associates, LLC
   Reston, VA

6. Grace Crowder
   NSA
   Fort Meade, MD

7. Louise Davidson
   National Geospatial Agency
   Bethesda, MD

8. Vincent J. DiMaria
   National Security Agency
   Fort Meade, MD

9. Rob Dobry
   NSA
   Fort Meade, MD

10. Jennifer Guild
    SPAWAR
    Charleston, SC

11. CDR Scott Heller
    SPAWAR
    Charleston, SC

12. Dr. Steven King
ODUSD
Washington, DC

13. Steve LaFountain
NSA
Fort Meade, MD

14. Dr. Greg Larson
IDA
Alexandria, VA

15. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

16. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

17. John Mildner
SPAWAR
Charleston, SC

18. Dr. Victor Piotrowski
National Science Foundation
Arlington, VA

19. Jim Roberts
Central Intelligence Agency
Reston, VA

20. Ed Schneider
IDA
Alexandria, VA

21. Mark Schneider
NSA
Fort Meade, MD

22. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

23. Ken Shotting
NSA
Fort Meade, MD

24. Dr. Ralph Wachter
ONR
Arlington, VA

25. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

26. Dr. Mark Gondree
Naval Postgraduate School
Monterey, CA

27. Professor Yeo Tat Soon
Temasek Defence Systems Institute
National University of Singapore
Singapore

28. Tan Lai Poh
Temasek Defence Systems Institute
National University of Singapore
Singapore

29. Ng Yeow Cheng
Defence Science & Technology Agency
Singapore